AGILE PLAYBOOK

Booz | Allen | Hamilton

Version 2.0 • June 2016

"Constantly think about how you could be doing things better. Keep questioning yourself."

- Elon Musk

Booz Allen

AGILE PLAYBOOK

TABLE OF CONTENTS

INTRODUCTION

Agile is the de facto way of delivering software today.

Compared to waterfall development, agile projects are far more likely to deliver on time, on budget, and having met the customer's need. Despite this broad adoption, industry standards remain elusive due to the nature of agility—there is no single set of best practices.

The purpose of this playbook is to educate new adopters of the agile mindset by curating many of the good practices that we've found work for teams at Booz Allen. As we offer our perspective on implementing agile in your context, we present many "plays"—use cases of agile practices that may work for you, and which together can help weave an overall approach for tighter delivery and more satisfied customers.

Core to our perspective are the following themes, which reverberate throughout this playbook.

We've come to these themes as software practitioners living in the trenches and delivering software on teams using increasingly modern methods, and in support of dozens of customers across the U.S. Government and the international commercial market.

- **Agile is a mindset.** We view agile as a mindset—defined by values, guided by principles, and manifested through emergent practices—and actively encourage industry to embrace this definition. Indeed, agile should not simply equate to delivering software in sprints or a handful of best practices you can read in a book. Rather, agile represents a way of thinking that embraces change, regular feedback, value-driven delivery, full-team collaboration, learning through discovery, and continuous improvement. Agile techniques cannot magically eliminate the challenges intrinsic to high-discovery software development. But, by focusing on continuous delivery of incremental value and shorter feedback cycles, they do expose these challenges as early as possible, while there is still time to correct for them. As agile practitioners, we embrace the innate mutability of software and harness that flexibility for the benefit of our customers and users. As you start a new project, or have an opportunity to retool an existing one, we urge you to lean to agile for its reduced risk and higher customer satisfaction.
- **Flexibility as the standard, with discipline and intention.** Booz Allen Systems Delivery uses a number of frameworks across projects, depending on client preferences and what fits best. We use Scrum, Kanban, waterfall, spiral, and the Scaled Agile Framework (SAFe), as well as hybrid approaches. But we embrace agile as our default approach, and Scrum specifically as our foundational method, if it fits the scope and nature of the work.
- **One team, multiple focuses.** Throughout this playbook, we explicitly acknowledge the symbiotic relationship between delivery (responsible for the "how") and value (responsible for the "what"), and we use terms like "delivery team" and "value team" to help us understand what each team member's focus may be. However, it's crucial to consider that, together, we are still one team, with one goal, and we seek a common path to reach that goal.

- **Work is done by teams. Teams are made of humans.** A team is the core of any agile organization. In a project of 2 people or 200, the work happens in teams. And at the core of teams are humans. Just as we seek to build products that delight the humans who use them, we seek to be happier, more connected, more productive humans at work.
- **As we move faster, we cannot sacrifice security.** According to the U.S. Digital Service, nearly 25% of visits to government websites are for nefarious purposes. As we lean toward rapid delivery and modern practice, we must stay security-minded. Security cannot be a phase-gate or an after-thought; we must bring that perspective into our whole team, our technology choices, and our engineering approach.

Who should use this playbook?

This playbook was written primarily for new adopters of agile practices, and it is intended to speak to managers, practitioners, and teams.

While initially written as a guide solely for Booz Allen Systems Delivery professionals, it is our hope that the community will also find value in our experience. We have deliberately minimized Booz Allen-specific "inside baseball" language wherever possible.

A Booz Allen internal addendum is also available for Booz Allen staff, with links and information only relevant for them. This is not because it is full of "secret sauce" proprietary information; rather, it is to keep the community version accessible and broadly valuable.

How should you use this playbook?

This playbook is not intended to be read as a narrative document. It is organized, at a high level, as follows:

1. **Agile Playbook context.** This is what you are reading now. We introduce the playbook, provide a high-level "Agile Model."
2. **The plays.** The plays are the meat of the playbook and are intended to be used as references. Plays describe valuable patterns that we believe agile teams should broadly consider—they are "the what." Within many plays, we describe techniques for putting them into practice. Plays are grouped into nine categories: Delivery, Value, Teams, Craftsmanship, Measurement, Management, Adaptation, Meetings, and Agile at scale.

Agile Playbook v2.0—What's new?

This is version 2.0 of our Agile Playbook. The first edition was published in 2013 and aided many practitioners in adopting and maturing their agile practice across our client deliveries and internal efforts. To address the regular changes happening in our industry, our firm, and the craft of software engineering, we wanted to update this material and reach an even wider audience. The following sections describe some specific changes you'll find in version 2.0.

Agile at scale, and DevOps

Two big shifts seem particularly apparent since we published the first Agile Playbook. The first is the growth of agile delivery on very large-scale projects. Our clients were commonly using agile approaches on efforts with fewer than 30 delivery staff, but it's now becoming a normal request with 50, 100, and 200 staff involved. With such size come some new quirks to keep delivery moving while living out agile values and principles at such scale. So we wanted to begin to tell that story.

The other large shift is the ubiquity of DevOps. No longer experimental, or just the domain of startups, DevOps' promise of faster delivery and higher quality is driving teams of all sizes and missions to adopt higher levels of automation and deeper team cohesion across disciplines that previously did not often integrate. This playbook is constructed with an eye toward aiding teams adopting a DevOps mindset and common practices while referencing our firm's DevOps Playbook.

Systems Delivery is bigger than it used to be!

In the last few years, not only has our business grown but also we acquired SPARC [Booz Allen Hamilton 2015] in October of 2015, which has been established as Booz Allen's Agile Hub in Charleston, SC! The SPARC team is known for its deep expertise in modern agile and DevOps methods, as well as its creativity in delivering solutions integrating cloud computing, UI/UX design, mobile technologies, cybersecurity, and production operation. This playbook represents a new collaboration of our Systems Delivery thought leaders and practitioners across Booz Allen, including SPARC.

Best learning practices

The first playbook referred to itself as a collection of best practices, but we want to clarify that. In most cases these are best *learning* practices. If you need a place to start or you want to understand something in context, we hope this playbook serves as a valuable guide and helps you keep walking toward high performance as a team or program. But, just as guitar virtuosos or Olympic skiers have moved past the form and rules they learned in their first few weeks of practice, we expect our delivery teams to learn the values here, start with the learning practices, and eventually innovate their way toward even higher performance methods that are unique to them.

Publicly accessible and kept up to date more continuously

Finally, we are excited to share that this playbook will move away from a monolithic release process, and it will be shared broadly. Now and in the future, you will be able to find (and contribute to) the latest version of the Agile Playbook through GitHub at https://github.com/booz-allen-hamilton/agile-playbook. If you are reading a printed copy of this playbook, we encourage you to also check it out on the Web for the absolute latest!

How and where can you contribute to this playbook?

We want to build this playbook as a community.

If you have ideas or experiences (or find a typo or broken link) you wish to share, we would love to hear about it. Contributions can be sent in one of two ways:

- Pull requests
- Email us at agile@bah.com—patch files preferred, but any feedback is welcome

We welcome feedback on the entire playbook, but we are looking for contributions in a few particular areas:

- New play or practice descriptions
- Reports or articles from the ground, completely attributed to you
- Favorite tools and software for inclusion in our tools compendium
- Additional references or further reading

Please be sure to take a look at our style guide before submitting feedback.

“Agile organizations view change as an opportunity, not a threat.

- Jim Highsmith

MEET YOUR GUIDES

The bulk of this content was developed by the following practitioners, coaches, and software developers from Booz Allen Hamilton.

We hope you will join us in this journey and engage us in collaboration and conversation.

Luke Lackrone
@lackrone

Timothy Meyers
@timothymeyers

Stephanie Sharpe
@Sharpneverdull

Claire Atwell
@twellLady

Lauren McLean
@SPARCedge

Camilo Gutierrez
@camilopga

Hallie Krauer
@SPARCedge

Kim Cumbie
@kimnc328

Marianne Rogers
@SPARCedge

Doug James
@SPARCedge

Noah McDaniel
@SPARCedge

We would like to also thank all of our reviewers, editors, contributors, and supporters:

Gary Labovich, Jeff Fossum, Dan Tucker, Kileen Harrison, Keith Tayloe, Elizabeth Buske, Joe Dodson, Emily Leung, Jennifer Attanasi, James Cyphers, Merland Halisky, Bob Williams, Gina Fisher, Aaron Bagby, and Elaine (Laney) Hass.

And, we would like to acknowledge the champions, contributors, and reviewers of the first version of this Agile Playbook: Philipp Albrecht, Tony Alletag, Maxim Aronin, Benjamin Bjorge, Wyatt Chaffee, Patrice Clark, Bill Faucette, Shawn Faunce, Allan Hering, Amit Kohli, Raisa Koshkin, Paul Margolin, Debbie McCoy, Erica McDowell, Johnny Mohseni, Robert Newcomb, Jimmy Pham, Rose Popovich, Melissa Reilly, Haluk Saker, Li Lian Smith, Alexander Stein, Tim Taylor, Loree Thompson, Elizabeth Wakefield, Gary Kent, Amy Dagliano, Alex Lyman, Alicia White, Kevin Schaaff, and Joshua Sullivan

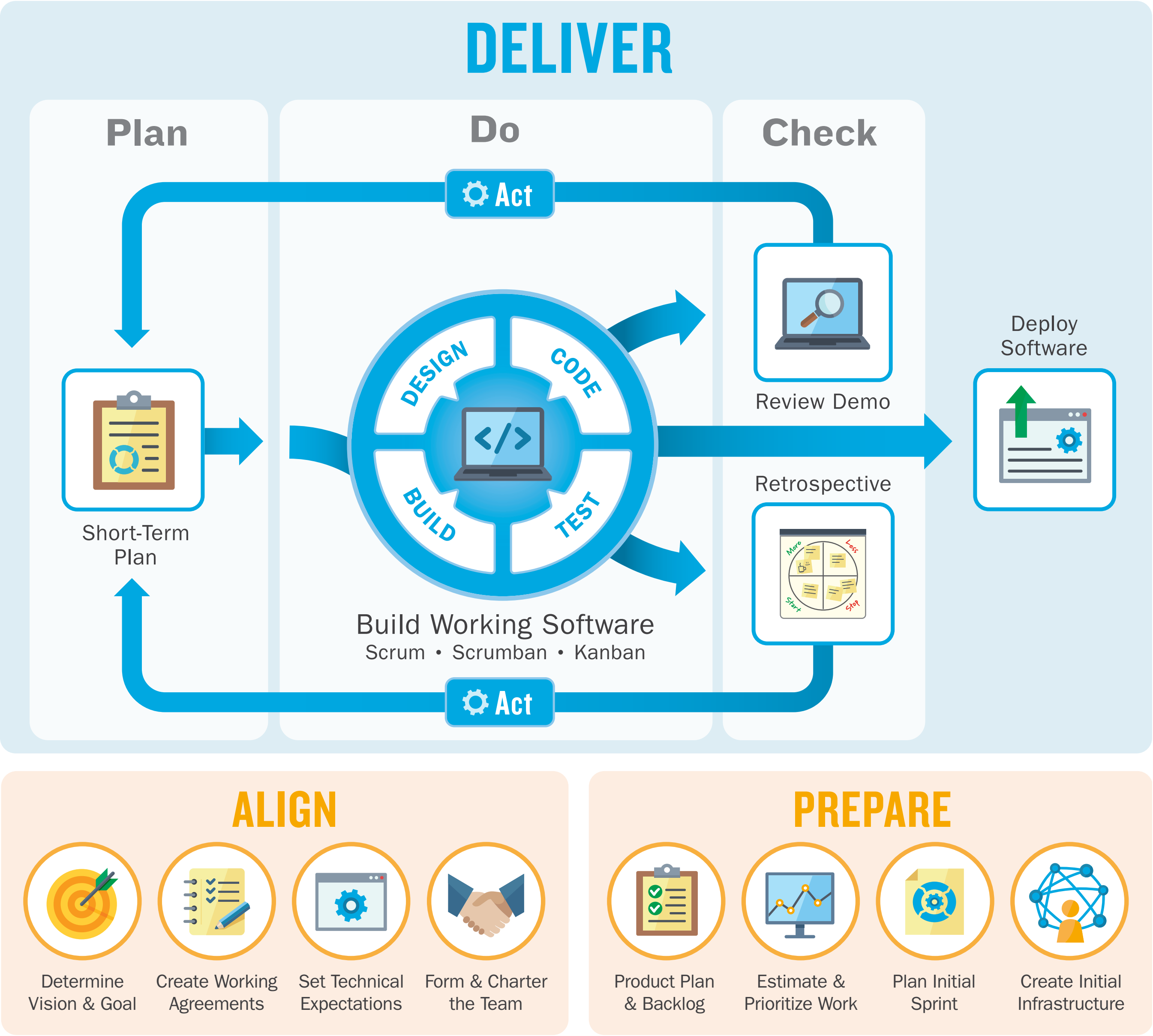
AN AGILE DELIVERY MODEL

Here we introduce the *agile delivery model* that we use to drive delivery across our business. Like most models, it is not perfect, but we believe it is useful. It illustrates the focus areas that a team may have over time—often simultaneously—and provides context for the plays and practices described in the rest of this playbook. Our intentions are to show that delivery is majority of our work and that *successful* delivery is built on a foundation of alignment and preparation.

To truly put this model into action, refer to the plays and practices.

We'll describe these focus areas in broad strokes. To truly put this model into action, refer to the details captured in the plays and practices.

Figure 1: Our agile delivery model



04.031.16_01
Booz Allen Hamilton

ALIGN

We must have a clear understanding of who we are and what we're trying to build.

With this focus, we cultivate relationships with our stakeholders and users. Together, we build a shared understanding of the project vision, goals, values, and expectations. We often employ chartering sessions to build product roadmaps and user personas, and to capture strategic themes.

We also create our team working agreements—how we will work together. These identify how we'll communicate, resolve conflict, and have fun, among other things, and we'll establish technical expectations, such as coding standards and our definitions of done.

This is a dominant focus area during a project's *first several days* (but no more! We want to align quickly and get to delivering as quickly as possible). While our teams begin with this focus, we realize that this is not only important during project startup. Teams may need to *realign* throughout the life of a project when significant change occurs.

PREPARE

Once we are clear on who we are and what we're here to do, the team needs to come together to get a look ahead, and prepare enough to get started.

With this focus, we build, estimate, and groom our product backlog. We put some thought into architecture. We sketch out a few sprints' worth of work—3 months or so. Here, we're trying to understand the things we will do soon. We accept that we are fuzzy on things we'll be doing a few months from now, and that time spent planning these things now is possibly waste. Because software development is primarily highly creative knowledge work, we have to do it to understand it. We continually discover. It is difficult to pull together a 12-month master schedule—if we did, it would probably be wrong in a matter of days. We embrace this truth instead of fighting it.

In addition to doing just-enough planning, we generally invest some time in our infrastructure and tooling. We want to make sure we can get from a commit to a build quickly. Let's smooth our deployment process so it's not a pain when time is tight.

DELIVER

This is where the rubber meets the road, so to speak.

With this focus, we transform the needs of users into valuable, tested, potentially shippable software. Generally, we follow small Plan-Do-Check-Act cycles. In the Delivery section of this playbook, we expand on several popular agile delivery frameworks, and when we use them.

When we are delivering, we build; we test; we keep designing; we keep talking to users. We inspect and adapt. We do this as much as we need to, until we are done. Any team members who are not actively delivering something for the current sprint are helping the team get ready for the next sprint.

“In my experience, there’s no such thing as luck.”

- Obi-Wan Kenobi

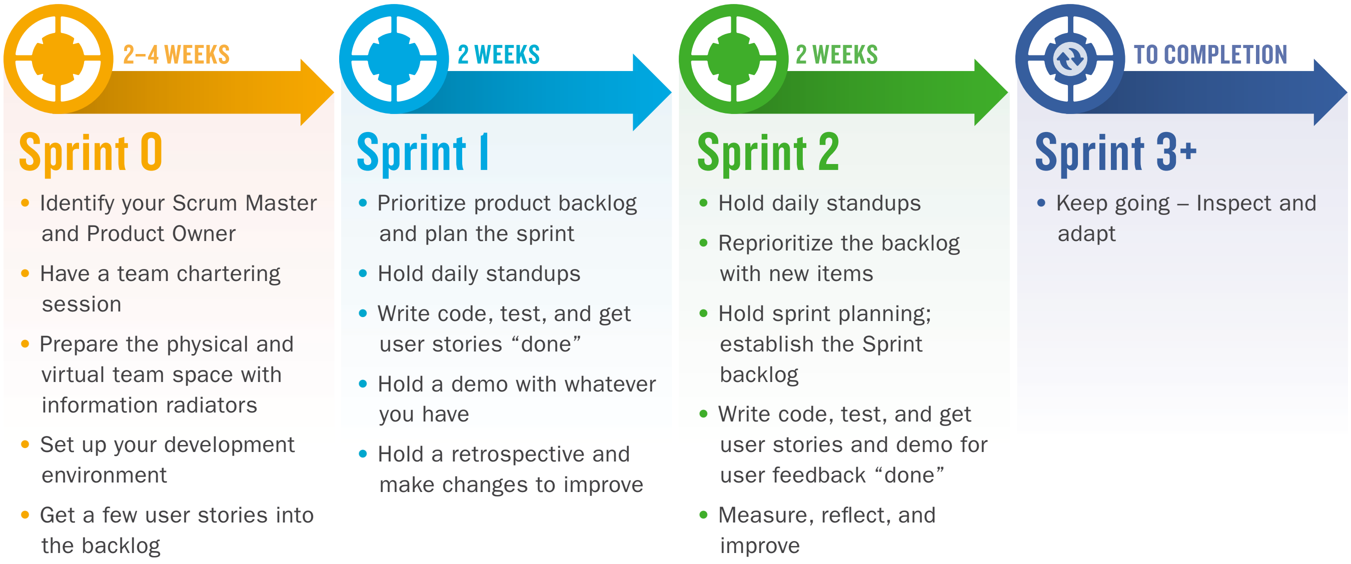
GETTING STARTED

So you've never been an agile team before. How to get started?

This guide gives a quick frame for getting started, tying together concepts you'll find more detail about in other areas of this playbook. Not all the terms are explained, so you'll want to reference the rest of the playbook to find more context and advice.

Keep in mind, it's not practical to totally change everything about how you work overnight. Start here; try these ideas and improve as you go.

Figure 2: Your first few months



2–4 WEEKS

Sprint 0

- Identify your Scrum Master and Product Owner
- Have a team chartering session
- Prepare the physical and virtual team space with information radiators
- Set up your development environment
- Get a few user stories into the backlog

2 WEEKS

Sprint 1

- Prioritize product backlog and plan the sprint
- Hold daily standups
- Write code, test, and get user stories "done"
- Hold a demo with whatever you have
- Hold a retrospective and make changes to improve

2 WEEKS

Sprint 2

- Hold daily standups
- Reprioritize the backlog with new items
- Hold sprint planning; establish the Sprint backlog
- Write code, test, and get user stories and demo for user feedback "done"
- Measure, reflect, and improve

TO COMPLETION

Sprint 3+

- Keep going – Inspect and adapt

04.031.16_02
Booz Allen Hamilton

2-4 WEEKS

Sprint 0

This period does not have to be strictly timeboxed; you want to get things in place so that you're ready to begin delivery as an agile team. Don't linger though—we want to be delivering!

Likely activities

- Identify your Scrum Master and Product Owner
- Identify the users and stakeholders
- Have a team chartering session
- Identify, define, and commit to an initial set of coding standards
- Identify initial architecture approach
- Identify some likely technologies
- Prepare the physical team space with information radiators
- Setup your development environment
- Set up your build infrastructure
- Set up a basic automated test environment
- Get a few user stories into the backlog
- Communicate the output of Sprint 0 to the teams and stakeholders

2 WEEKS

Sprint 1

Likely activities

- Hold daily standups
- Prioritize product backlog
- Estimate top user stories
- Establish a sprint backlog
- Write code; test; get user stories "done"
- Hold a demo with whatever you have
- Hold a retrospective and make changes to improve

2 WEEKS

Sprint 2

Likely activities

- Hold daily standups
- Reprioritize the backlog with new items
- Hold sprint planning, establish the sprint backlog
- Write code; test; get user stories “done”
- Hold a demo; gather user feedback
- Measure your velocity
- Reflect and identify improvements

TO COMPLETION

Sprint 3+

Likely activities

- Keep going—inspect and adapt!

“If, on your team, everyone’s input is not encouraged, valued, and welcome, why call it a team?”

- Woody Williams

THE PLAYS

Plays describe valuable patterns that we believe agile teams should broadly consider.

These plays and practices are intended to be used as reference. Within many plays, we describe practices that teams can do to turn the plays into action.

DELIVERY

This section describes how agile teams work together to produce value that satisfies their stakeholders.

VALUE

This section describes how agile teams can understand the value of the work they do.

TEAMS

Agile teams are where the work gets done. Team members care about each other, their work, and their stakeholders. And agile teams are constantly stretching, reaching for high performance. This section describes plays for team formation, organization, and cohesion.

CRAFTSMANSHIP

This section walks through practical ways to inject technical health into your solutions.

MEASUREMENT

Measurement affects the entire team. It is an essential aspect of planning, committing, communicating, improving, and, most importantly, delivering.

MANAGEMENT

Where is the manager on an agile team? This section explores how the manager leads in an agile organization.

ADAPTATION

This section looks at ways to regularly examine and find ways to improve the team and product.

MEETINGS

This section describes common meetings for agile teams, and how to effectively use your time together.

AGILE AT SCALE

This section describes some of our initial thoughts on scaling agility.

DELIVERY

Agile teams are biased to action and are constantly seeking ways to deliver more product and more often.

This section describes how agile teams work together to produce value that satisfies their stakeholders:

> ***Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.***
>
> ***Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.***
>
> ***Simplicity—the art of maximizing the amount of work not done—is essential.***

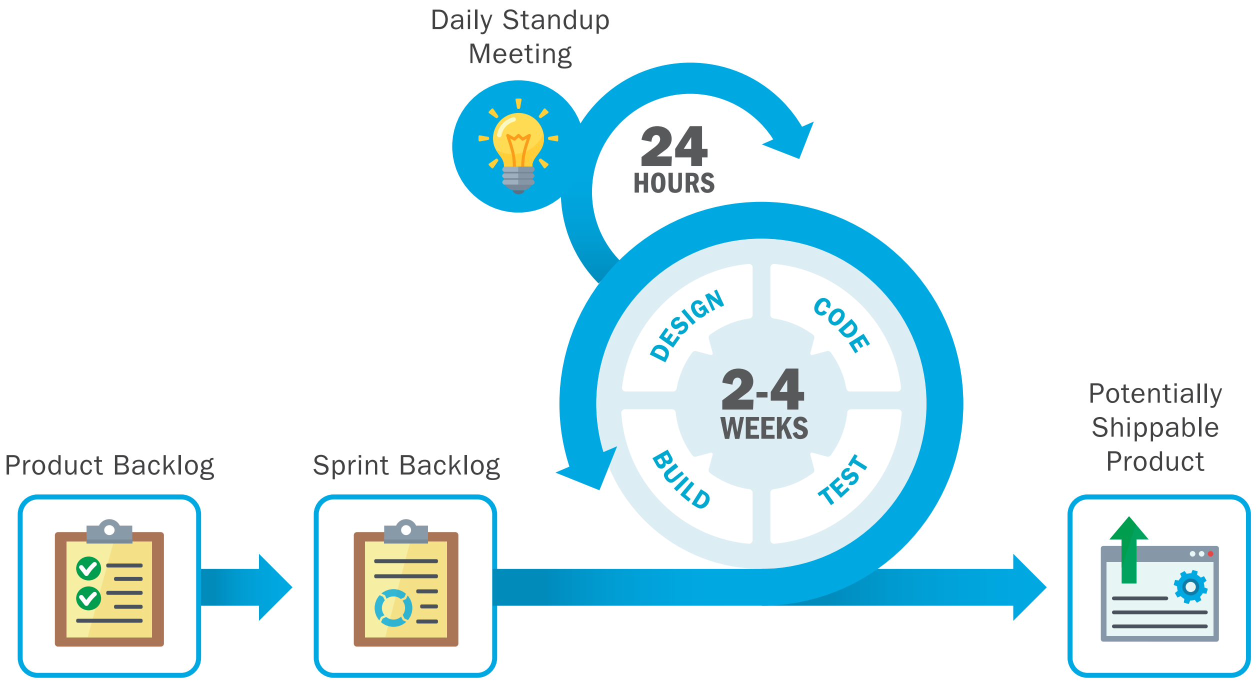
Play: Start with Scrum

Start with Scrum for agile delivery, but with an eye for "agility."

Scrum is the most popular delivery framework for agile teams to use, by far; so much so that it's often confused for "agile" itself. Scrum is a powerful, lightweight product delivery framework that has existed for 25 years. The definitive description of the framework is maintained by its creators in the Scrum Guides [Sutherland and Schwaber 2013]. Because the Scrum Guides are such well-maintained and well-used resources, we won't try to explain everything about Scrum here.

Scrum was developed to rapidly deliver value while accommodating the changes that are inevitable in product delivery. It's also meant to create a predictable pace for the team.

Figure 3: The Scrum framework



04.031.16_03
Booz Allen Hamilton

Traditionally, a product is designed, then developed, then demonstrated or released to the customer. This occurs mostly as a sequence and over long stretches of time. Often, the customer is unhappy with the result and wants changes. Since we are so late in development, changes found after release are typically very costly. Scrum, however, incorporates frequent demos and feedback to mitigate surprise requirement changes. All the project work gets cut into short development iterations known as sprints. Scrum emphasizes that work planned in sprints must be small, well understood, and prioritized.

Each sprint is typically 1–4 weeks long (and stays consistent for a given team). During a sprint, the delivery team chooses the high-value work it can complete; the team focuses on just that work for the sprint's duration. At the end of each sprint, the team demonstrates the working software it produced. During the demo, the team gathers feedback that helps shape the direction of the product going forward.

Practically, this means that design decisions are not made way ahead of time, but rather right before or even during active development. Instead of having heavy, top-down design, design emerges and evolves over several iterations: develop, demo, gather feedback, incorporate feedback, develop, demo…

Over the course of several sprints, a picture of progress and direction emerges. Customers and management are kept informed through progress charts (see Measurement section) and end-of-sprint demos. It becomes easy to keep everyone informed while avoiding many pitfalls of micromanagement.

If Scrum is followed, many of the traditional problems associated with complex projects are avoided. The frequent feedback prevents projects from spending too much time going in the wrong direction. Furthermore, because of Scrum's iterative nature, projects can be terminated early (by choice or circumstance) and deliver value to the end user.

Iteration—trying something and looking at it—is core to how Scrum operates.

Scrum is built on three pillars: transparency, inspection, and adaptation

Transparency

Many of the challenges teams face boil down to communication issues. Scrum values keeping communication and progress out in the open; doing things as a team; being transparent. The fact that Scrum's ceremonies (Sprint Planning, Daily Standup, Sprint Review, and Retrospective) are intended for the whole team speaks to the importance of transparency.

Inspection

Inspecting things is how we know if they're working. Everything on a Scrum team is open to inspection, from the product to our process.

Adaptation

As we inspect things, if we think we would benefit from doing it differently, let's try it! We can always adjust again later. Notably, Scrum's Sprint Review ceremony gives us an explicit, regular opportunity to adapt based on how the product is coming along; the Retrospective ceremony does the same for how our team is working.

A variety of development teams use Scrum, from those working on highly complex systems with an unknown end, through operations and maintenance patching. The key to using this method is ensuring the maintenance of a groomed backlog and allowing for the flexibility needed in this short learning cycle.

Play: Seeing success but need more flexibility? Move on to Scrumban

If Scrum is too restrictive or there are too many changing priorities within a sprint, consider Scrumban to provide the structure of ceremonies with the flexibility of delivery.

⊘ < No process >

In environments where we are used to doing work in our own swim lane, or a single person possesses most the knowledge, it's easy to skip defining processes. The team needs to find ways to work together which often take the form of a process. Repeatable well-understood processes for regularly occurring tasks help the team move faster, reduce stress, and integrate new team members. A process needs to be understood by the entire team and perhaps documented for it to work. The repeatable tasks are often defined during team chartering and revisited at retrospectives. Some common processes to reflect on: Who checks our work? When and how do we deploy our software? How do we avoid becoming single threaded on a capability? How do we track our work? Do we need a tool? What is our defect tracking process?

Scrumban is derived from Scrum and Kanban (described below, in the next play) as the name would suggest. It keeps the underlying Scrum ceremonies while introducing the flow theory of Kanban.

Scrumban was developed in 2010 by Corey Ladas to move teams from Scrum which is a good starting point in agile, to Kanban, which enables flow for delivery on demand [Ladas 2010]. Kanban focuses on flow, but it does not have prescribed meetings and roles—so we borrow those from Scrum in this Scrumban model. The primary difference versus Scrum is that the sprint timebox no longer applies to delivery in Scrumban.

Instead, the team is constantly prioritizing and finishing things as soon as possible. In Scrumban, we keep Scrum's cadence just to have our ceremonies so we don't miss out on planning together, showing our work, and having a time for reflection.

A strict work-in-progress limit (WIP limit) is set to enable team members to pull work on demand, but not so many things that they have trouble finishing the work at hand. Scrumban has evolved some of its own practices.

Examples of unique practices to Scrumban include the following:

- *Bucket-size planning* was developed to enable long-term planning where a work item goes from the idea bucket, to a goal bucket, then to the story bucket. The story bucket holds items ready to be considered during an on-demand planning session.
- *On-demand planning* moves away from planning on a regular cadence, instead only holding planning sessions when more work is needed. Items to be pulled into the Kanban board are prioritized, finalized, and added to the Kanban board.

Scrumban is useful for teams who are very familiar with their technical domain and may have constantly changing priorities (e.g., a team working on the same product for an extended amount of time). The flexibility of Scrumban allows for the backlog to be re-prioritized quickly and release the product on demand.

Play: If you are ready to kick off the training wheels, try Kanban

Try Kanban on only the most disciplined teams and when throughput is paramount.

Kanban is a framework adopted from industrial engineering. It was developed to be mindful of organizational change management, which is apparent in the four original principles:

- Start with existing process.
- Agree to pursue incremental, evolutionary change.
- Respect the current process, roles, responsibilities and title.
- Leadership at all levels.

So, in Kanban, you will not (inherently) be receiving a bunch of new titles, or using much new vocabulary.

In 2010 David Anderson elaborated with four “Open Kanban” practices tailored for software delivery:

- Visualize the workflow.
- Limit WIP.
- Manage flow.
- Make process policies explicit.
- Improve collaboratively (using models and the scientific method).

In Kanban, you fundamentally want to make all of your work visible, continuously prioritize it, and always flow things to “done.” This is great for a software team that issues several new releases per week, or per day. A pitfall, however, is that if priorities are allowed to change too often, no work will ever get done. So, be mindful about finishing things and not starting too many things.

Kanban is appropriate for teams ready to self-regulate, rather than rely on timeboxes. The practices require discipline to enable flow. An operations and maintenance team with a small backlog could benefit from Kanban, as it would enable delivery of small items as needed and ensure all issues are getting to a done state. In addition, mature agile teams with a highly automated pipeline could use Kanban as a way to enable quick flow of value to production.

VALUE

Most software has more features than necessary.

Agile teams emphasize prioritizing features by the value they bring to real users and stakeholders. Considering the value of things is just as important as delivering working software, since time spent on non-valuable features is wasted time.

Play: Share a vision inside and outside your team

> “***Business people and developers must work together daily throughout the project.***”

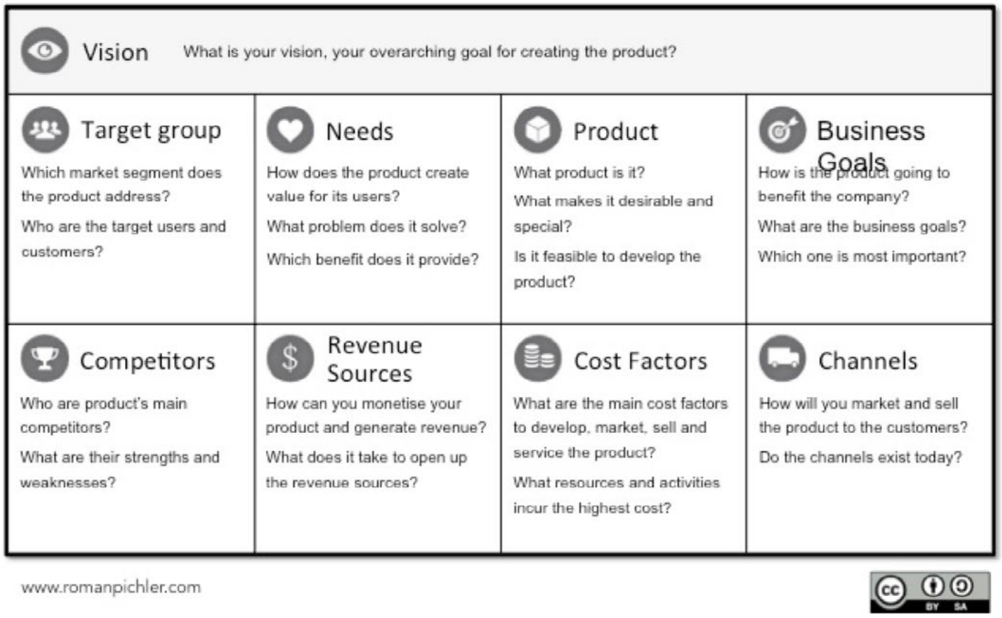
The vision is the foundation upon which product decisions are made. When at critical junctures, turn to the vision to help determine which direction will help the vision become a material reality. At the team and individual levels, the vision provides a common mission to rally around and helps understand the long-term goals, as well as incremental goals.

Practice: Product Vision Statement

The vision for the project should be encapsulated in a product vision statement created by the Product Owner. Akin to an “elevator pitch” or quick summary, the goal of the product vision statement is to communicate the value that the software will add to the organization. It should be clear and direct enough to be understood by every level of the effort, including project stakeholders. The Vision Board by Roman Pichler is a nice, simple template for forming this statement [Pichler Consulting 2016].

Once you have your vision board, work with the Product Owner and key stakeholders to test the vision with the target group to see how well it resonates with eventual users.

Figure 4: Product Vision Board by Roman Pichler [Pichler Consulting 2016]



Vision What is your vision, your overarching goal for creating the product?

Target group Which market segment does the product address? Who are the target users and customers?	Needs How does the product create value for its users? What problem does it solve? Which benefit does it provide?	Product What product is it? What makes it desirable and special? Is it feasible to develop the product?	Business Goals How is the product going to benefit the company? What are the business goals? Which one is most important?
Competitors Who are product's main competitors? What are their strengths and weaknesses?	Revenue Sources How can you monetise your product and generate revenue? What does it take to open up the revenue sources?	Cost Factors What are the main cost factors to develop, market, sell and service the product? What resources and activities incur the highest cost?	Channels How will you market and sell the product to the customers? Do the channels exist today?

www.romanpichler.com

Practice: Product box

The product box is another way to try to crack into the product's vision. You might try this exercise as an alternative to working with the Vision Board. The product box is a great way to engage a whole team in the conversation around the vision and value of the project at hand, and to have some fun together while doing so. While there are many versions of this idea, Innovation Games is a well-known one. As described there, "[Ask your stakeholders] to imagine that they're selling your product at a tradeshow, retail outlet, or public market. Give them a few cardboard boxes and ask them to literally design a product box that they would buy. The box should have the key marketing slogans that they find interesting" [Innovation Games 2015]. We have also seen this work nicely by imagining that your product is appearing in the App Store; what would the description, icon, screenshots, and reviews look like?

Once you have your product box, use focus groups or hallway testing with your target group to discover how well the vision resonates with target users and to understand their expectations for what's inside.

Practice: Product roadmap

Most teams need a *product roadmap* to understand high-level objectives and direction for the project. We think of this as the project's North Star. If we've deviated distinctly from it, we should have a good reason, and we probably need to update the roadmap. Be sure to include the ultimate project goals in the roadmap, keeping in mind their value added and the desired outcomes from the customer's point of view. The roadmap should loosely encapsulate the overall vision and give a sense for when capabilities will be delivered or intersecting milestones are going to occur. The roadmap should probably cover the next 6–12 months, and only in broad strokes. On the roadmap, releases contain less detail the farther they are into the future. Your team will have to talk through rough sizing of work and prioritization during the creation process. **You're not building a schedule;** you're trying to paint a *plausible* picture. Be sure to build this together as a team, or at least review and revise it together. Too many roadmaps are built by leadership and never have buy-in from the team.

Practice: Release plan

Once the roadmap is complete, a *release plan* may be created for the first release. Each release should begin with the creation of a release plan specific to the goals and priorities for that release. This plan ensures that the value being added to the project is consistently reviewed and, if necessary, realigned to maximize the overall value and efficiency of development. Like the product roadmap, the release plan should include a high-level timeline of the progression of development, specific to the priorities for that release only. The highest value items should be released first, allowing the stakeholders visibility into the progression of the work.

User engagement is often overlooked when developing release plans. We find that the best roadmaps and release plans include user communication and engagement strategies, such as training and rollout.

Play: Backlogs—Turn vision into action

> “ ***Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.*** ”

Once we have and share our vision, we understand the big stuff, but we have to turn this to action quickly. A core practice for agile teams is to have a *backlog* of work. In this context, a backlog is a prioritized set of all the desired work (that we know about) we want to do on the project.

Backlog basics

Treat your backlog as a “catch all”; any item that moves the team ahead to a final product or project goal can be added to your backlog. New features, defects, abandoned refactoring, meetings, and other work are all game to be placed there. Additionally, keep in mind that your backlog will evolve in detail and priority through engagement with the end user. Your backlog should be a living, breathing testament to your product, as you will iteratively refine your backlog as your build your product. Product backlog items are added and refined until all valuable features of that product are delivered, which may occur through multiple releases during the project lifecycle.

Backlog creation

A backlog is made of epics and user stories. User stories are simply something a user wants, and they’re sized such that we understand them well; epics are bigger than that and we need to break them down further so the team can actually execute. Generally, anyone can add something to the backlog, but the Product Owner “owns” that backlog overall—setting the priority of things and deciding what we really should be working on next.

Good backlogs follow the acronym DEEP

Detailed

The backlog should be detailed enough so that everyone understands the need (not just the person who wrote it).

Estimated

The user story should be sufficient for the delivery team to provide an estimated effort for implementing it. (Stories near the top of the product backlog can be estimated more accurately than those near the bottom.)

Emergent

The product backlog should contain those stories that are considered emergent—reflecting current, pressing, or realistic needs.

Prioritized

The product backlog should be prioritized so everyone understands which stories are most important now and require implementation soon.

We dive further into both epics and user stories in the following play.

Practice: Product and sprint backlogs

A *product backlog* is a prioritized list of product requirements (probably called user stories), regularly maintained, prioritized, and estimated using a scale based on story points. It represents all of the work we may want to do for the project, and it changes often. We have not committed to deliver the full scope captured in the product backlog.

A *sprint backlog* is a detailed list of all the work we've committed to doing in the current sprint—just a few weeks of work. Once we set it up in sprint planning, it remains locked for the duration of the sprint. No new (surprise!) work should be added. The whole team should keep it up to date by (at least daily), and items should be marked complete based on the team's agreed definition of done.

Play: Build for and get feedback from real users

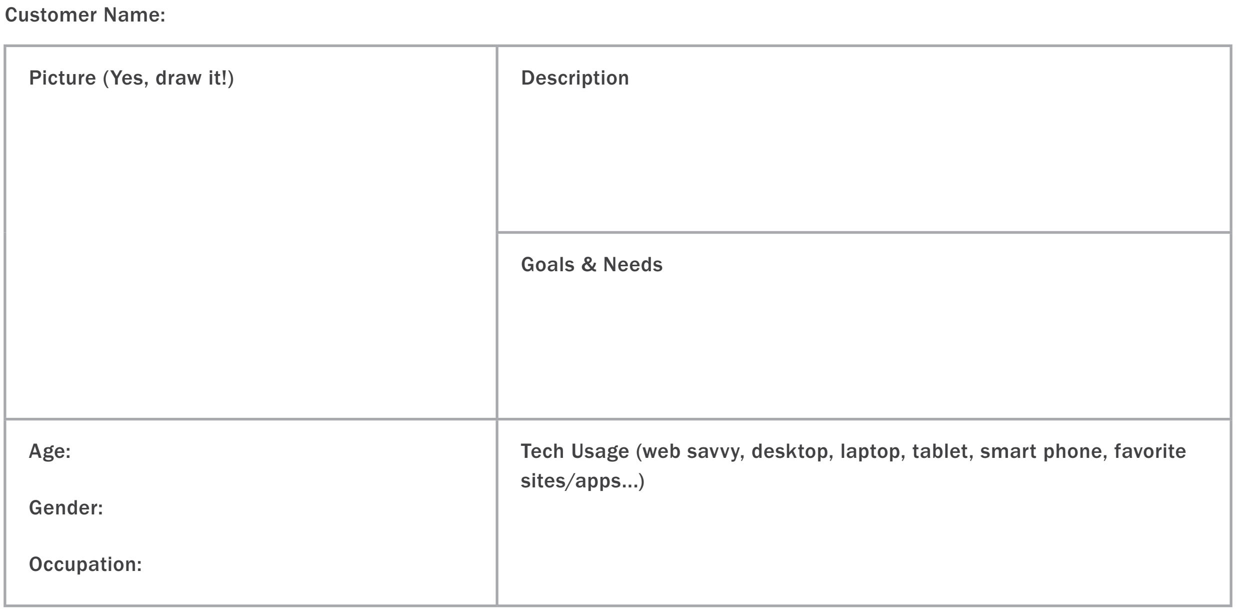
> “***Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.***”

Practice: Define personas

Personas are fictitious people who represent the needs of your users, and they help us understand if our work is going to be valuable for the people we're trying to reach. They can be very useful at the start of the requirements gathering process, but typically remain important throughout.

Each persona should capture the user and their individual needs. Create a template with an area to draw a picture of the user and separate spaces to describe the user personally, but also describe their desired goals, use cases, and desires for the software.

Figure 5: A persona template



Customer Name:

Picture (Yes, draw it!)	**Description**
	Goals & Needs
Age: **Gender:** **Occupation:**	**Tech Usage (web savvy, desktop, laptop, tablet, smart phone, favorite sites/apps...)**

Practice: Talk with users about needs, not solutions

Consider the difference between "I want the software to have one button for each department in my business" and "I need to be able to access each department in my business from the software."

Users often think they know exactly what they want to see, but we find it can be much more effective if we understand the needs and then exercise creativity in how to provide a delightful experience that satisfies that need.

As you can, guide your users into conversations of value and need, and let the delivery team work through the solution.

Practice: Epics and user stories

A template for user stories and epics

Epics and user stories are described below, but both share a typical template:

- **Title (short)**
- **Value statement**
  - Outlines and communicates the work to be completed and what value delivering this epic or story will bring to a specific persona/user
  - Format: As a (user role/persona) I need (some capability), so that (some value)
- **Acceptance criteria**
  - An outlined list of granular criteria that must be met in order for the story or epic to be fully delivered and adequately tested and verified. This helps inform development and understanding of when the story is ready to demonstrate or test further

User stories

User stories are the agile response to requirements, and you can call them requirements if you like.

They often look like this:

- As a solo traveler
- I want to safely discover other travelers who are traveling alone
  - So I can meet possible companions on my next trip.

There are a few things that are different about user stories versus typical requirements.

- They are communicated in terms of value, from the user perspective.
- They might look a little lightweight at first. We agree we need to *tell stories*—we know there are details that can only be sussed out through collaborating with our users and stakeholders. We understand that what is written down is imperfect. What we want to do is capture enough to get started!

The Product Owner collaborates with the delivery team to develop user stories that will mold the product's functionality. User stories are identified and prepared throughout the project's lifecycle.

There are two devices we typically use to describe good user stories: the three Cs and INVEST.

The three Cs

Card

This is the description of the user story itself, written on a card or in a tracking tool. The card should give us enough detail to get started and know whom to talk to.

Conversation

We acknowledge that anyone implementing needs to—and should—speak with some of the players involved in the value that story will deliver. So they need to go have a conversation and record anything that needs to be preserved from that conversation.

Confirmation

Once implemented, every user story needs to be verified. We call this the Confirmation. And we should record in the user story how we intend to verify it. This could be the list of acceptance criteria, test plans, and so on.important now and require implementation soon.

INVEST

A way to see if your user stories are pretty good is to consider the INVEST acronym.

***I**ndependent*

Stories should be as independent as possible, so they can be implemented out of order.

***N**egotiable*

We should be able to discuss the details of the user story, find the optimal solution, and not treat the initial writing as gospel.

***V**aluable*

A story must deliver value to the user or customer when complete.

***E**stimable*

Stories should be such that we can estimate their effort.

***S**mall*

User stories should be small enough to prioritize, work on, and test. For teams using sprints, user stories should be able to be completed inside one sprint.

***T**estable*

We should know how to verify and test the story.

Epics

Epics are broad functionalities we want our product to deliver. They are larger than user stories. Epics would typically take multiple sprints to deliver, and would ultimately be broken down into many user stories. You can still write epics like user stories, and the guidelines above typically apply, with the exception of Small and Estimable.

Strong v. weak user stories

Table 1: Strong v. weak user stories

Strong User Stories Are...	Weak User Stories Are...
✓ Developed and prioritized by the Product Owner	✗ Created with limited Product Owner involvement
✓ Written from the user's perspective	✗ Developed without designating the specific user or user group that will receive value from the story
✓ Simple and concise, with clear alignment to business value	✗ Missing a description of the business value
✓ Entry points to a conversation on how the implementation activities can be decomposed	✗ Technical specifications that don't link to the user's point of view
✓ Written with easy-to-understand acceptance criteria	✗ Open ended with no means to validate acceptance

TEAMS

Agile teams are where the work gets done.

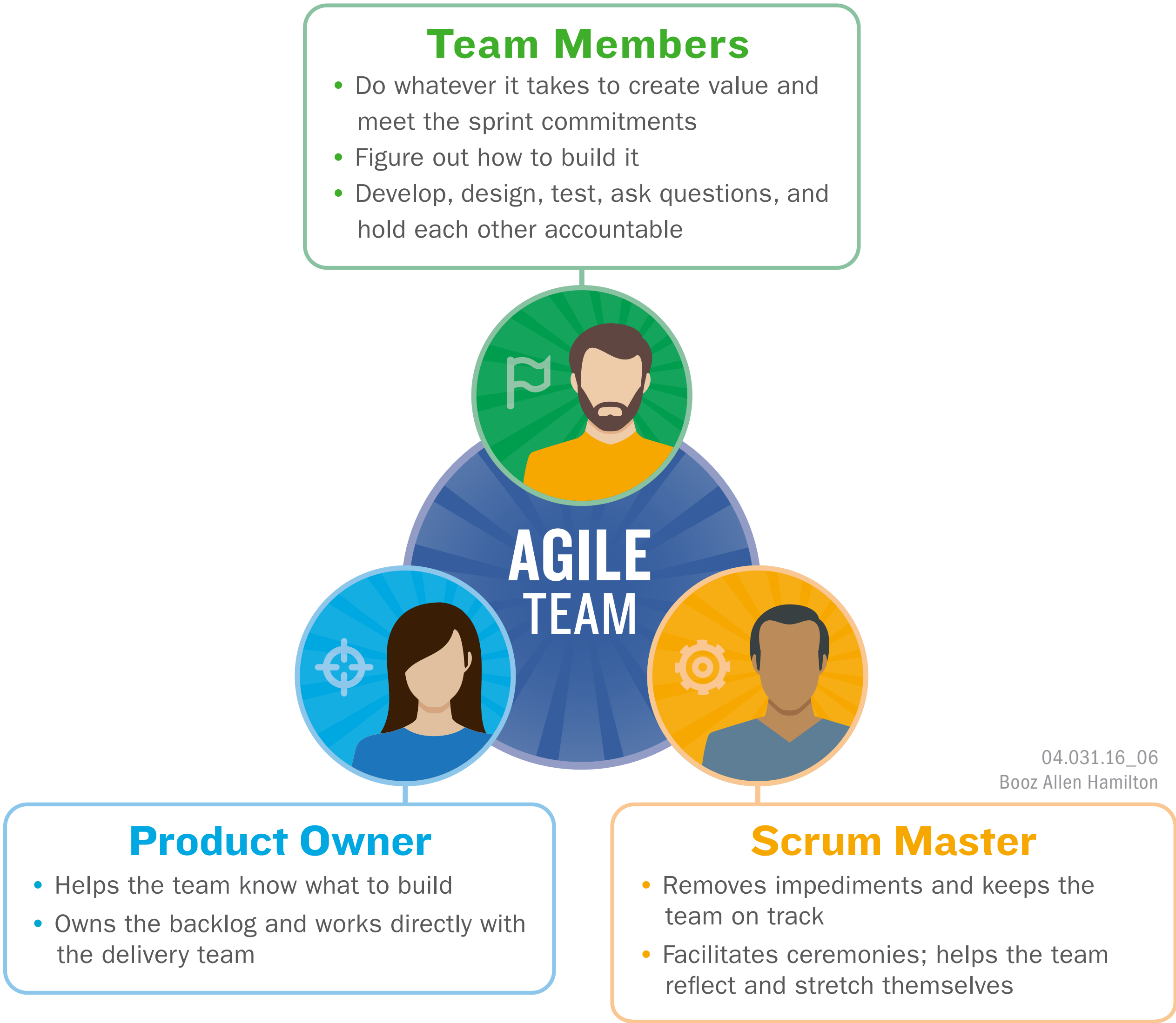
Team members care about each other, their work, and their stakeholders. And agile teams are constantly stretching, reaching for high performance.

> “ ***Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.***
>
> ***The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*** ”

Play: Organize as Scrum teams

Agile teams are cross-functional in nature and work together to analyze, design, and build the solution their customers need. Agile team members, together, can understand the business or mission needs and create an effective solution that meets those needs.

Figure 6: An agile team



Team Members

- Do whatever it takes to create value and meet the sprint commitments
- Figure out how to build it
- Develop, design, test, ask questions, and hold each other accountable

Product Owner

- Helps the team know what to build
- Owns the backlog and works directly with the delivery team

Scrum Master

- Removes impediments and keeps the team on track
- Facilitates ceremonies; helps the team reflect and stretch themselves

04.031.16_06
Booz Allen Hamilton

We use one particular set of vocabulary here for this playbook, which is reflective of some of the most common terms found across government and industry; but, individual client environments may dictate different names for things. We stress that the jobs and structures mentioned here are helpful in any context, and we urge teams to strive for the greatest agility possible, then seek to continuously improve through small changes.

Practice: Build the Scrum team

Scrum gives us a simple model for a team, and we believe this is a valuable frame for most agile teams. Scrum suggests just three roles:

- Scrum Master
- Product Owner
- Team Member

The Scrum Master

The Scrum Master is an experienced agilist, responsible for upholding agile values and principles for the delivery effort. The Scrum Master helps the team execute its agreed-upon delivery process and typically facilitates common team ceremonies, like daily standup meetings, planning meetings, demos, retrospectives, and so on. The Scrum Master biases the team toward action and delivery and stretches the team to continuously improve, hold each other accountable, and take ownership of the way the process works.

Although Scrum is a specific agile framework, (explained more in the Delivery section), the notion of a delivery team facilitator in an agile team is so common and so effective, we feel it's simplest to refer to this person as a Scrum Master—even for a non-Scrum team—given it is the most common term for that role.

Scrum Masters typically:

1. Make sure the right people are talking together, all the time
2. Remove barriers for the team
3. Reflect progress back to the team
4. Shield the team from interruptions
5. Ensure the team's process is followed, including facilitating team ceremonies
6. Coach and mentor the team and customers on living out agile values and principles
7. Tune into team morale, energy, and conflict and address issues
8. Ensure the team has an environment that fosters productivity, openness, and teamwork

Growing Scrum Masters

We routinely encounter clients and teams who place more stock in entry-level certifications, like Certified Scrum Master, than is truly warranted. While this certification and others like it provide a great overview of the Scrum framework, it does not magically make someone an effective Scrum Master. Because of this, we recommend the following learning plan for our Scrum Masters:

Table 2: Scrum Master learning plan

Core or Elective	Certification	Description
Core	ICP	ICAgile Certified Professional
Core	PSM I	Professional Scrum Master I
Core	ICP-ATF	ICAgile Certified Professional in Agile Team Facilitation
Elective	PMI-ACP	PMI-Agile Certified Practitioner
Elective	ICP-ACC	ICAgile Certified Professional in Agile Coaching
Elective	SA	SAFe Agilist

The Product Owner

The Product Owner is the person who most considers the mission or business value of the solution being developed. They are responsible for maximizing the return on investment for the development effort, and they speak for the interests of the users. They could be from the client organization; if they are from the Booz Allen team, they represent the client's perspective. They interact regularly with the delivery team, clarifying needs and providing feedback on designs, prototypes, and iterations of the solution.

Product Owners typically:

1. Create and maintain the product backlog
2. Prioritize and sequence the backlog according to business value
3. Assist with elaboration of epics into user stories that are granular enough to be completed soon (like a single sprint)
4. Convey the vision and goals for the project, for every release, and for every sprint
5. Represent and engage the customer
6. Participate in regular team ceremonies (like standups, planning, reviews, retrospectives)
7. Inspect progress every sprint, accept or reject work, and explain why
8. Steer the team's direction at sprint boundaries
9. Communicate status externally
10. Terminate a sprint when drastic change is required

Growing Product Owners

We use the following learning plan to grow Product Owners and agile business analysts:

Table 3: Product Owner and agile business analyst learning plan

Core or Elective	Certification	Description
Core	ICP	ICAgile Certified Professional
Core	PSPO I	Professional Scrum Product Owner I
Core	ICP-BVA	ICAgile Certified Professional in Business Value Analysis
Elective	PMI-ACP	PMI-Agile Certified Practitioner
Elective	ICP-ATF	ICAgile Certified Professional in Agile Team Facilitation
Elective	SPM/PO	SAFe Product Manager/Product Owner

The team member

Team members are, of course, the other members of the team, such as testers, developers, and designers. They are the people who, collectively, work together to deliver value on the project. They carry a diverse set of skills and expertise, but they are happy to help out in areas that are not their specialty. Team members, together, take collective responsibility for the total solution, rather than having a “just my job” outlook. Team members do whatever they can to help the team meet its sprint goal and build a successful product.

On functional roles and T-shaped people

On agile teams, there should be less emphasis on being a “tester” or a “business analyst” or a “developer”; we should be working together, sharing the load, collectively getting to the goal. That said, we still value the specialties and disciplines our team members bring to the project. Agile teams are ideally made of **generalizing specialists,** sometimes referred to as **T-shaped people,** who collaborate across skill sets but bring valuable depth in a useful specialty to the project. This means that while a team member may have deep knowledge in a particular area, known as a specialist, they also need to build knowledge broadly, known as a generalist.

⊘ *< Assuming a class is enough >*

While agile education is a core piece of the adoption puzzle, sending a few people (or worse, a single would-be Scrum Master) to a 2- or 3-day certification course is not a recipe for success. In many ways, agile represents a paradigm shift for individuals, teams, and leadership. These entry-level agile certifications simply introduce this new way of thinking and some of the popular frameworks or practices. They do not equip an individual with the essential change management skills required to achieve sustainable agility. We employ a team of certified and experienced agile coaches to partner with teams on their journeys towards agility and high-performance. Read more about creating a coaching capability from Lyssa Adkins [Adkins 2015].

The “T” in T-shaped is made up of a vertical line representing the deep knowledge of a specialist and a horizontal line representing the broad knowledge of a generalist. Specialists who are deep but not broad can only accomplish work in a particular area and can become a bottleneck when they are the only team member who can accomplish the work. Conversely, generalists who have broad but not deep knowledge can become a bottleneck when the work requires a deeper understanding or greater skill set. Teams of generalizing specialists build trust in one another by collectively committing to goals, sharing knowledge, actively mentoring, and delivering solutions.

On bigger teams, it's reasonable that people may live in their specialty more than on a smaller team. Team composition is heavily influenced by financial feasibility. Teams working for a client under contract often have constraints related to a labor category and hourly rate. This is the perfect opportunity to bring junior members along and encourage learning skills outside the specialty as needed. For example, a junior developer may benefit from learning test automation, which is adjacent to development. Constraints on skill sets or level are the perfect opportunity to emphasize the need for teams with T-shaped members. Ultimately this is a balance that the team, with all its local context, should talk about and adapt through inspection and conversation.

⊘ < *Assuming delivery or maintenance are someone else's problem* >

So we're back to throwing things over the fence in a "that's not my job" mentality (see generalizing specialist). Not only does this cause tension between teams but it also leads to poor quality and just plain "bad stuff." When we don't take responsibility for delivering valuable, quality solutions as a team in the larger sense of the word, we short change our customer.

On team size

Our experience shows that a cross-functional team of less than 10 is the preferred team size. This is supported by the Project Management Institute and generally follows the Scrum Guides. This size would include a dedicated Scrum Master and a Product Owner. As the project size scales beyond 10, the effectiveness of the team per person declines, and significantly more time is spent coordinating work. If the scale of the work requires a large team, we need to think about how that can be divided into multiple cross-functional teams that are tightly coordinated but loosely coupled.

More thoughts on scale can be found in the Agile at Scale section.

Team structure patterns

In its simplest frame, agile teams have people who focus on the value of things—what needs to be built—and other folks focused on the delivery of things—how we will build it, what's possible. Though they operate as a single team with one goal and one purpose, you can also consider that there are two virtual, smaller teams here: a value team and a delivery team. There are a few patterns we see work well, so we explain those in the next three plays.

⊘ < *Scrum Master is also the Product Owner* >

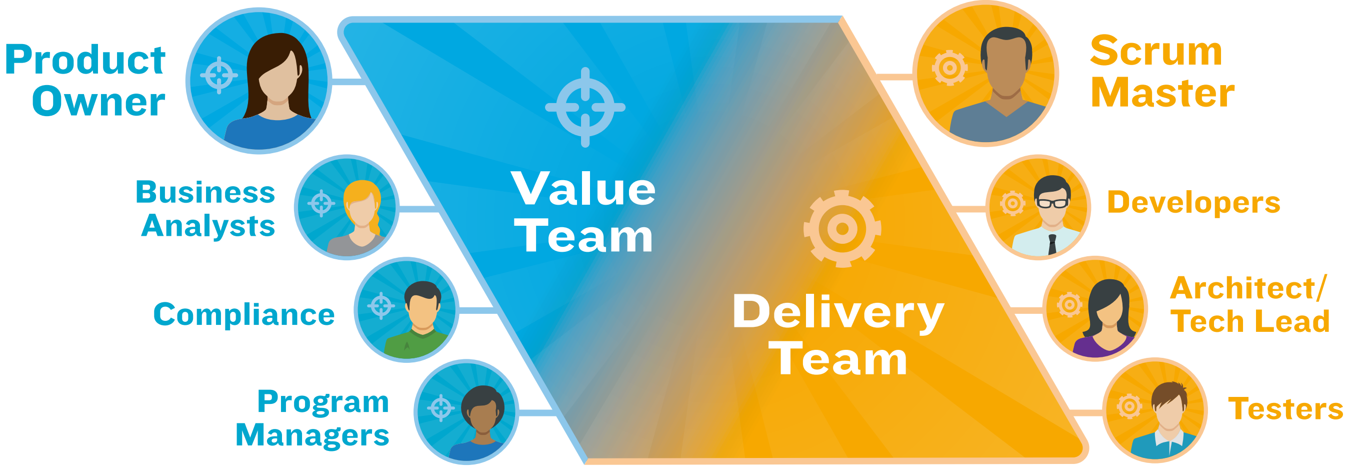
The Product Owner naturally wants as much value completed in the shortest time to market. The Scrum Master is there to facilitate the team, help unblock impediments, and stay attuned to the team needs. The natural tension that exists with the Scrum Master's and Product Owner's goals helps the team find balance. If they are combined into one person, you lose the benefits of each role. Either the Product Owner no longer pushes the team to get the most, based on their Scrum Master persona; or the Scrum Master no longer cares about the team's pulse and pushes as the Product Owner. The team needs both roles.

Play: Expand to a value team when one Product Owner isn't enough

If the mission needs are sufficiently complex or there are complicated relationships with multiple client stakeholders, it might be impossible to have just one Product Owner. In that case, we recommend thinking of a larger value team. This team might grow to around 10 people and would typically include representatives like business analysts, compliance interests, end users, and so on.

Figure 7: An agile project team

AGILE
Project Teams



▼ Value Team

Communicates and represents client needs by **defining priorities and acknowledging acceptance**

▲ Delivery Team

Cross-functional group that does whatever it takes to produce a **valuable, working product**

04.031.16_07
Booz Allen Hamilton

In this case, there can still be a Product Owner, but that person is now the value team's facilitator, bringing together all those perspectives and creating a common voice, and typically would not be able to trump the other stakeholders. In this setting, the jobs of the Scrum Master and Product Owner become more difficult—trying to coordinate all the interests at play—but they can still be very successful.

Play: Scale to multiple delivery teams and value teams when needed

If the overall solution scope is very large, the expertise required is sufficiently diverse, or the timeline is constrained such that a single team is insufficient to produce the solution, then you'll have to scale up to multiple delivery teams, each with its own Product Owner/ value team. There are a few frameworks that help in scaling agile, which we discuss later in this playbook. Additional roles are likely required, like architects to keep the technology sufficiently robust and coordinated.

⊘ *< Scrum Master is also the Project Manager >*

The Scrum Master is there to support the team and uphold their process, whereas the Project Manager is there as an interface with the client and keep the project on the rails. Much like the Product Owner, the Project Manager should be in a natural tension with the Scrum Master. Managing the work often means tracking resources, budget, risk, tasking, and cross team dependencies, and ensuring delivery on the work committed. The Scrum Master focuses on the team needs. By combining these roles, you make both less effective. The Project Manager taking the Scrum Master stance will put the team first, being less likely to push the team on delivery. The Scrum Master taking the Project Manager stance would be less likely to protect the team from the outside pressures, and impose less favorable team atmosphere.

Play: Invite security into the team

The security perspective needs to be ever present in the process, not just an after thought. From establishing requirements, through designing the system and implementing features, to operations and sustainment, security needs to be considered and baked in. In a modern team, it is everyone's responsibility to think about, address, and implement secure practices. Security should be embedded in the culture; it isn't just a step at the end, or "that other team" down the hall. It will typically make sense for security-focused professionals to find their home in the value team, when we think about how the software may need to attain a certain accreditation or certification. But, security-mindedness is essential for the delivery team as well, because we want those practices and habits that build secure software to be part of the routine work.

⊘ < People are rewarded for non-agile, non-collaborative behavior >

The hero approach is often applauded by organizations because it shows immediate results and it's easy to identify the reason for success. In the long run, it diminishes the importance of tackling issues as a team. When each team member strives to take on everything alone, they risk burnout, becoming a bottleneck for the team, and stunting growth of team members. When team members are rewarded for collaborative behavior, they build trust, grow skill sets, exchange knowledge, mentor one another, and share responsibility in success and failure.

Play: Build a cohesive team

We strongly believe in keeping the team together. And the simplest team is under 10 people, has a dedicated single Product Owner, and has a dedicated single Scrum Master. This team is committed to just one project at a time and can plan and estimate together. The longer a team is together, it tends to be more predictable, has great potential for high-performance, and really enjoys working together.

Practice: Chartering

At the beginning of a project, or after significant change on a project (in scope or in team makeup), we recommend team chartering. This is a meeting, ideally in person, together—even for a team that's otherwise distributed. And the real focus of this meeting is: Who are we, and what are we doing? We recommend working through activities to get to know each other and find out what each member's skills and passions are. How do you like to have fun? How do you like to communicate? What makes you happy? What makes you frustrated? You'll likely want a good facilitator to pull this chartering meeting together. Teams find it has lasting effects on the sense of community, empathy for each other, and overall effectiveness.

⊘ < Moving people to work, rather than work to teams >

Management made switches to resources on an organization chart, so the work is dispersed and assigned to the new project. Managers should consider how to foster high-performing teams. Teams that work together for a stretch of time begin to find their stride and form an identity. When a team member is moved to "put out fires" on another project, the team members left behind must reform. They must establish velocity, revisit working agreements, and begin to normalize around their new composition. This disruption ends up being costly since the receiving team must do the same with the addition of the new team member. When the project needs to increase capacity, the additional stories should be prioritized into the team's backlog instead of moving individuals.

Team Charter

Team members

- Who are the team members? Names and preferred contact info (email, call, text, etc.)
- Who will be acting as the Product Owner (or representative) and Scrum Master?

Collaboration locations

- Team area location
- Conference call/video chat info
- Agile board name & URL
- Wiki URL

Working together

- What does our team value? What do we stand for?
- Working agreement: How will we get work done and stay happy along the way? (e.g., How will absences be communicated? How will we hold ourselves accountable for tasks/action items?)
- How will our team handle conflict?
- Core working hours: Do you have core hours when team members will be available/ reachable

Product Owner

- Who is the Product Owner?
- Who are our stakeholders and how do they coordinate with the Product Owner?
- Product Owner availability: When is the Product Owner available/unavailable? Does the Product Owner sit with the team? Which ceremonies does the Product Owner lead/attend?
- What do we do when the Product Owner is unavailable for agile ceremonies?

Sprint cadence

- What day does our sprint begin and end? (recommend not Monday or Friday)
- Sprint planning: time/day/location
- Daily standup: time/location
- Sprint retrospective: time/day/location
- Backlog refinement/grooming: time/day/location
- Who is invited to each ceremony? (list attendees)
- Important known milestones: Are there any dates or deliveries that we know of already? Begin to build out our team roadmap (in Confluence or some other accessible place) using those known milestones

continued on next page >

Coordinating with other teams/vendors

- Do we ever need to coordinate across teams?
- How will our team handle this situation?
- How will we proactively manage dependencies/blockers/etc.?
- How will you collaborate? Who will facilitate the coordination?

Definitions

- Definition of done: How will we define done on our team? Getting on the same page about this and having the discussion up front and being able to refer back to it will save the team later. Revisit what done means to you once in a while to make sure you update it as things change (e.g., passed unit tests, documentation done, peer reviewed, code checked in…)
- Definition of ready: How can we make sure something is ready to be worked or ready to be planned? (e.g., meets INVEST criteria; independent, negotiable, valuable, estimable, small, testable)
- Product vision: It's important to know where you are going as a team. As a pod as part of the larger team, what is the vision for the larger team? What is the vision for your particular project/group?
- Estimation: What estimation scale/points are we using? Will we estimate using planning poker?

Practice: Colocate

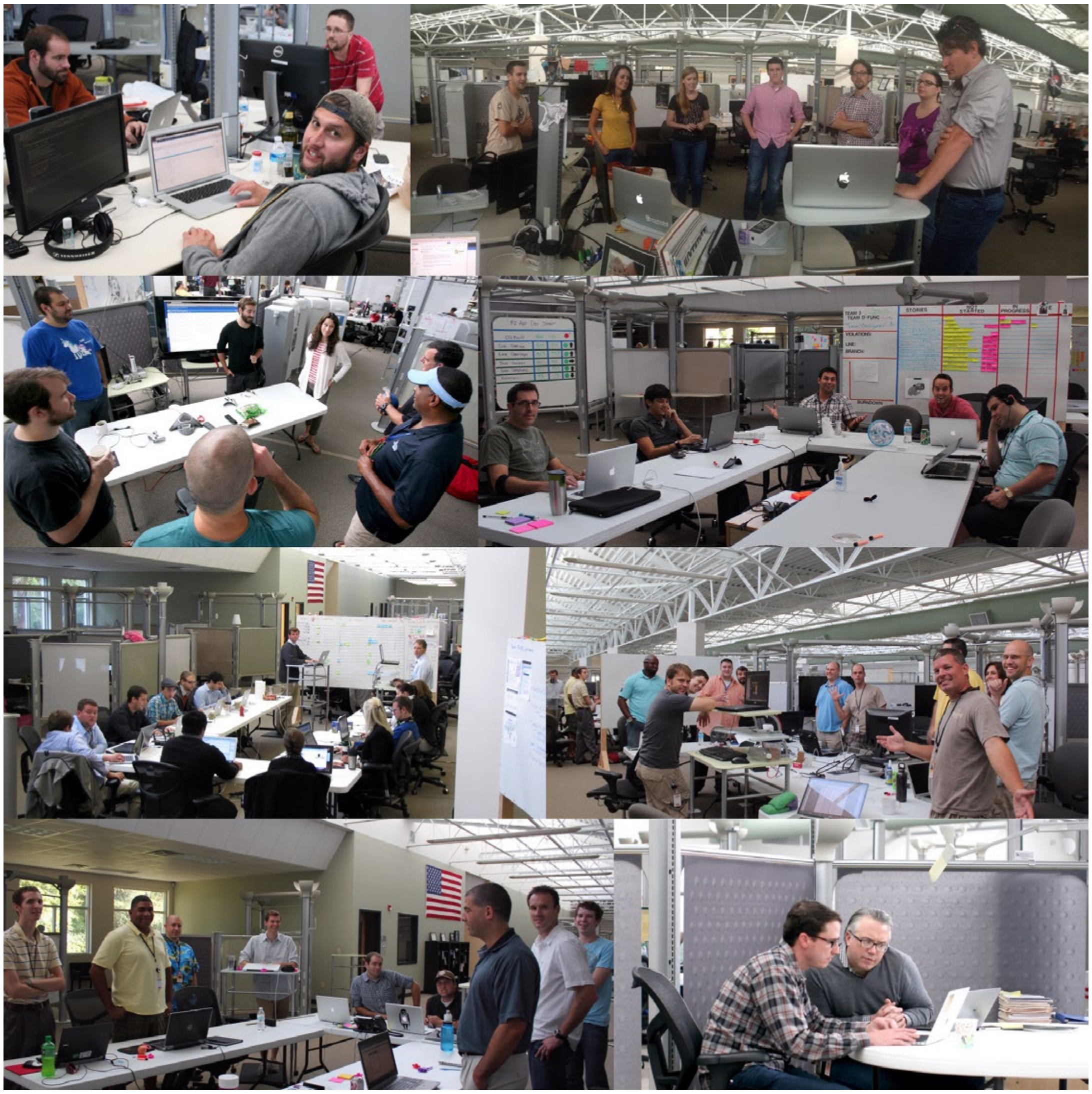
Colocation is still best

We still recommend colocating agile teams as much as possible. Being together in one room, or in close proximity, allows a lot of things that are much tougher otherwise: You can pull together an impromptu meeting or demo more easily. You can put an idea on a whiteboard, or tape up some designs on the wall and get feedback quickly. And there's no substitute for hearing your teammates' sighs, squeals, and applause, signaling you to what's happening on the project right this moment. Our brains have evolved to communicate with much more than words, and seeing each other's faces is just as important as the language being used.

⊘ < Matrixed resources >

Matrixing resources is common in organizations but can be detrimental to the individual and the team cohesion. One key to happy employees is the understanding of responsibilities, priorities, and where employees fit in the grand scheme of the organization. Individuals on teams need to be able to commit their time and be fully engaged in the work. Additionally, one team means one set of priorities. The cost of context switching between teams or responsibilities means the person will likely be less effective at either one.

Figure 8: Booz Allen's Charleston agile delivery hub



A great team room contains a few things

- Comfortable workspaces for each team member
- Enough space to move around and pair with another team member at their desk
- Information radiators on the walls that reflect project status and monitoring
- A comfortable space to lounge and be social or to use for meetings
- A private space for team members to conduct sensitive conversations or high-concentration work

Practice: Distribute with intent

Distribution can work

Colocation isn't always possible. For lots of different reasons, we may lean to distributed teams: availability of talent, cost, client geographies, and so on. We might be fully distributed, or just have a few team members in satellite locations. When we are distributed, we have to put in extra effort to stay communicative and engaged with each other, and every common team meeting requires more work from the facilitators. Planning meetings, standups, retrospectives—distribution complicates how we collaborate for all of them. Clearly, technology plays a role here to keep us together—tools like chat, video, electronic whiteboards. If your team is distributed, you need to approach this intentionally; fewer good things will happen by accident than what we see in colocated teams.

We have a few tips that can help distributed teams

- **Use video.** Make sure each team member, regardless of location, has a camera, and create a virtual team room by having everyone on camera all the time. This way, when speaking on the phone or chatting via IM, we still get to see facial expressions.
- **Get off email.** Consider banning email for communication inside your team. Emails are easy to lose, and waiting for email responses slows teams down. Use the phone more, and get a persistent chat room tool for your team.
- **Facilitate from the lowest-bandwidth perspective.** If you need to lead a meeting, and four people will be in a conference room together and two will be attending via Skype, then you should get on Skype and conduct the meeting from another room, treating each person as remote. This ensures no one is overlooked.
- **Talk about it.** There's no substitute for regularly asking each other how things are going, and if we can improve. As time goes on, you'll need different rhythms, different tools, or just to express what's frustrating you about the way we work. Create space for those conversations. Distribution can be tough, but there are many teams that do it well, and it unlocks a certain freedom that those team members really enjoy.

CRAFTSMANSHIP

Agile software engineering is still software engineering. Ron Jeffries tells us, “The software we build has to be robust enough to support the business need. It needs to be sufficiently free of defects to be usable and desirable. It needs to be well structured enough to allow us to sustain its development as long as required.

Robust, reliable, well designed. These are not things that we just automatically get by having a Product Owner and a Scrum Master. Much less are these things we get by having really good Portfolio Vision and an Agile Release Train.

“If we do not build it well, all our teamwork, communication, retrospectives, business focus and WIP limitation are for nothing.” -Ron Jeffries

Delivering great systems requires a dedication to the craft and an eye toward excellence. Technical agility and strength are necessary for teams to be truly agile. This section walks through practical ways to inject technical health into your solutions.

Play: Build in quality

> ***“Continuous attention to technical excellence and good design enhances agility.”***

We will delve into some ways to ensure quality is always considered in every aspect of software delivery. Security and quality cannot be thought of as bolt-on or follow-on functions after development is done. Building security and technical excellence into the solution as we go is a shared responsibility, and we need the team to continue to stretch itself to achieve better results over time. The team will do this by using agreed-upon practices, talking together, regularly inspecting, and automating their work.

Practice: Technical definition of done

In addition to the Product Owner accepting user stories as complete, the team must also determine what practices they will follow to determine when the user story is of sufficient quality to review with the Product Owner. This definition of done must be explicitly discussed and followed by all team members. The definition often includes the practices within this play.

An example of “done” might look like this:

1. Code complies with the standards agreed upon through manual or automated static code inspection
2. Unit test has been written and passed
3. Code has been reviewed by a peer
4. Code has been checked in to a repo
5. Code has passed automated or manual security and/or compliance inspection
6. Code has been successfully integrated into a build
7. User story has been successfully deployed to a test environment
8. User story has passed functional testing

Practice: Adopt a coding standard

Forming a **coding standard** is an essential task for any agile team. Creating agreed-upon documentation that defines the style in which code will be written and any practices to be followed or deemed unacceptable prevents the occurrence of potential problems. The team's coding standards also likely will include topics such as how errors are handled and how code is structured (directory convention, etc.). The intent is to ensure the delivery team develops code uniformly, which aids future code updates and developer comprehension and eases code review. Coding best practices fall into two groups: independent best practices (e.g., variable naming conventions) and dependent best practices (e.g., how to use aspect-oriented programming principles).

⊘ ***< Poor technical practice >***

Agile does not dictate technical practices, but often it highlights issues that exist. Increasing the pace of deployments or automation shows weaknesses in the technical practices through a faster learning cycle. The goal is to decrease time dedicated to defect correction or manual processes that can slow down a team.

The team and its senior developers should define the initial coding best practices the team will use during software development activities. These practices should be presented to the team and discussed in detail to ensure complete understanding of how to execute acceptable coding practices, as well as the implications of not doing so. During retrospectives, these practices should be considered and modified as needed to reflect possible improvements for the team.

Practice: Paired work

Pairing with a colleague benefits the team through early identification of potential issues, shortened review cycles, and enhanced commitment to the quality of the product, in addition to being a learning opportunity for both parties. It should be employed when creating any artifact for the project, from code through documentation.

Practice: Code reviews

There are two primary benefits of **code reviews.** First, any bug found during review is cheaper to fix then (by orders of magnitude) than if it is found later in the process. Second, a team that is practiced in inspecting the code tends to be able to embrace the agile principle of collective code ownership. All of the code (and any bugs identified) is the responsibility of the team, rather than a specific individual. This mindset results in a tighter, higher quality product.

Agile teams use code reviews to identify and fix bugs and weaknesses that may have been overlooked. Code reviews also enable senior developers to mentor junior developers on how to write better quality code.

Code reviews are performed for all code changes and should be included in the team's software development workflow. Generally, code is not included in the next build or made available to the test team unless the team has held an review on that code. Typically, the following elements will be included in the team's code review practice:

- Review changed code
- Verify functionality

⊘ ***< No documentation >***

Agile is not an excuse to toss out all of your documentation! The Manifesto simply states, "Working software over comprehensive documentation." But, we do emphasize looking at your documentation and understanding its value proposition, just as we do for features we build. Who is asking for it? Why are they asking for it? What will it be used for? If you are on a team that is not documenting anything, it is worth investigating.

- Review results of security and code quality scans.
- Review test case results, including any automated unit or regression tests.

Teams should create a brief training for new developers to introduce them to how reviews work on their team.

For code reviews

Do…

- Look for code style issues
- Look for obvious coding mistakes
- Ask questions/pose alternatives if code appears complex
- Look for code areas that seem fragile
- Provide constructive feedback, contributing to a solution if possible
- Look for concise and clear comments to explain unusual cases, techniques, and anything that feels non-obvious
- Be respectful in your comments—picking apart someone else's code can make the author feel vulnerable
- Ensure someone has the job to shepherd the review through timely completion
- Ensure team members have had a chance to review code before considering the review done
- Link the code review artifacts with how you track the original work request (ticket, requirement, etc.)

Don't…

- Close out a review someone is actively working on
- Try to obtain a 100% complete understanding of all the code
- Examine every possible input case; this also takes way too much time
- Expect overly verbose comments

Collaborative code reviewing tools, such as Atlassian's Crucible, provide the ability for inline comments for code with informative feedback. Additionally, look for ways to automate code quality basic review using review tools such as Atlassian Clover or Code Climate.

Practice: Source code analysis

Source code analysis is empirically proven to be one of the most effective pre-test defect-prevention techniques, increasing quality and reducing downstream rework. We required more advanced methods to address defects, vulnerabilities, and sub-standard coding practices to ensure the highest levels of structural quality, maintainability, and security before application deployment. Project teams needed an automated and repeatable way to measure and improve the application software quality of multiplatform, multi-language, and multi-sourced applications. Implementing source code analysis includes:

- Planning for and incorporating code scanning as part of the continuous integration activities to have a real-time characterization of application health at the code level with tools such as SonarQube
- Performing structural quality scan on the code with tools such as CAST; using scan results as a way to direct development efforts due to specific vulnerabilities or business priorities

⊘ *< No definition of done >*

When a team has not established and agreed to a definition of done, there can be confusion. Each team member has a differing opinion of what “done” means. Whether it’s applied to stories, features, epics, sprint, release, ready for production, etc., the definition of done should be established by the team prior to the work being accomplished. This is usually done during team chartering but can be established at any time prior to planning the work. This level setting across the team means members can hold one another accountable when it comes time to deliver. This also prevents the lingering story that never seems to be done.

Practice: Secure coding

Secure coding is a practice to prevent known vulnerabilities and keep the code secure by refactoring as new vulnerabilities are identified, or as the environment (operating systems, browsers, web specifications, etc.) changes. Implementing this practice includes:

- Defining secure coding practices at the start of the project
- Reviewing practices during the code review for the checked-in file
- Using free open-source software that checks against defined best practices (general coding practices and custom practices that the team defines in the tool); see the Tools section of this document for more information on code review and scanning tools
- Integrating automated code reviews into code check-in and continuous integration to gain the best results with minimal manual effort
- Scanning applications for vulnerabilities on the server. Three types of scans would benefit the team:
- Scan the server for container-based vulnerabilities. The Retina network security scanner is one of the best tools to use for this purpose; another tool for the same purpose is the Nessus Vulnerability scanner
- Perform static analysis on the code with tools such as IBM Appscan or HP Fortify
- Perform automated penetration testing using tools such as Hailstorm
- Documenting the actions to be taken for each type of scan

Secure coding best practices incorporate the Open Web Application Security Project (OWASP) Enterprise Security Application Programming Interface (ESAPI) to simplify and standardize the implementation of security functions in the environment, unless environmental factors prohibit deployment. OWASP ESAPI toolkits help software developers guard against security-related design and implementation flaws by ensuring simple, strong security controls are available to every developer. An integrity check of software products is included to facilitate organizational verification of software integrity after delivery.

⊘ *< Not enough attention to quality >*

Agile is sometimes only seen as the ability to deliver faster, and that becomes the only focus. By building in multiple layers of checks and testing, less time correcting defects is needed in the long run. Build quality check practices into your processes, and look for opportunities to automate them. This will ensure fewer defects and less time fixing those defects. The cost of a defect found early in the lifecycle is significantly cheaper to fix than in production.

Practice: Address security through the entire stack

From physical location to client-facing application, teams must be versed in the skill sets needed to ensure applications are secure. All talent from development to security professionals should be security-minded: trained in software development practices that are secure throughout. This includes awareness of threats and attack vectors not only in the layer of application being built but also in the surrounding layers; layers belonging to partners, shared libraries, and so on. Further, as security concerns span the entire application, an approach that only addresses one layer in the stack is especially at risk for breaches in any of the other layers. Defense-in-depth is an architectural security pattern well suited for modern applications, which employs a multi-layered approach to security. As a team, practice continual learning and diligence in understanding your technology stack and its security posture.

As a starting point, regularly consider these elements:

1. Are security tools used to check software vulnerabilities, and can we decide on an action for each vulnerability?
2. Are security scans included as a part of each automated build, and is that security posture radiated to the team and stakeholders?
3. Have compliance requirements, such as NIST, RMF, 800-53, been addressed as technical stories?
4. Does your security strategy also address containers, network, firewalls, and operating system for vulnerabilities? As an example, Netflix's Security Monkey is a tool that checks the security configuration of your cloud implementation on AWS.
5. Have functional security test scripts been developed and executed to verify security features, such as authorization, authentication, field level validation, and PII compliance?
6. Does the configuration of security components, such as the perimeter firewall and Intrusion Detection/Prevention System (IDS/IPS), follow a similar model in terms of provisioning and configuration as application servers? Use Infrastructure as Code artifacts, to describe these configurations and to ensure the ability to consistently and repeatedly configure components, prevent system administration drift, and support audits and traceability of changes.
7. Is advanced network monitoring in place to actively find vulnerabilities or active attacks?
8. Is security talent embedded within teams, and is each team member from developer to security professional security-minded? Remember, security is a shared concern.
9. Is the process of defining, implementing, and monitoring security, from beginning to end, an iterative cycle throughout the life of the software? This is a proven strategy as illustrated by the U.S. Postal Service's Secure Digital Solutions Electronic Postmark Identity Proofing project.
10. Are software security fundamentals implemented, such as OWASP's Top 10, as well as project-specific security concerns, such as HIPAA or PII compliance?

Practice: Refactor your solutions

Code refactoring is a technique for restructuring an existing body of code—altering its internal structure without changing its external behavior. We refactor to improve the nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity to support maintenance. There are two other benefits to refactoring:

- **Maintainability.** It is easier to fix bugs/defects when the source code has been written so that it is easy to understand and grasp. This might be achieved by reducing large routines into a set of individually concise, well-named, single-purpose methods. It might also be achieved by moving a method to a more appropriate class or by removing misleading comments.
- **Extensibility.** It is easier to extend the capabilities of an application if it uses recognizable design patterns and provides some flexibility where it may not have existed previously.

⊘ **< No testing >**

If you have a project that seems straightforward or you are very familiar with the technology, it may seem like testing is just going to slow you down. A key tenet of the Manifesto is "working software." Whatever we deliver has to work, and we have to know that it works. Testing may just confirm what you already know, but it is part of technical excellence and is a foundation of quality.

Practice: Invest in unit testing

Unit tests are a foundational software engineering practice for development teams. This practice refers to developers writing small tests that test individual components in the system, typically with code that's automatically run during builds. Designing unit testing is a challenging task for the development team. The unit test infrastructure and architecture need to be designed during Sprint 0 of the project. The unit tests could cover all layers of the application or target only certain layers.

Unit tests should be included in the definition of done for any work item, and developers should discuss plausible test strategies when the work is planned. When a developer makes the code ready for the sprint's releasable software, this indicates that the unit tests are also ready.

Play: Build in quality ... and check again

Testing incremental functionalities of the product developed by the agile team involves reviewing the user stories to ensure they meet the definition of done, are considered complete, and have passed the acceptance testing criteria. Preparing for testing activities includes any artifacts required to successfully execute testing, such as the scripts, code, and data. The objective of any testing activity is to determine whether the incremental product developed satisfies the intended requirements and also proves to be a testable component. Along with functional and regression testing activities, automated testing and acceptance testing are also performed.

Practice: Use test-driven development (TDD)

TDD is a software development practice that increases code quality by ensuring high unit test coverage. Unit test coverage has been proven to increase overall code quality by providing the first level of test on which later testing continues to check additional quality factors. When using TDD, developers write a unit test first, then produce only enough code to pass that test, then refactor the code to elegantly integrate into the existing codebase. When diligently followed, this practice builds the foundation for a robust, tested body of code.

TDD is performed whenever code is written. It is **not a testing methodology;** it is a software development technique. The objective is 100% coverage for all software with automation.

The system's correct behavior is well-defined in the body of tests that developers can confidently make changes and deploy. These tests augment any code documentation by demonstrating the functionality, enabling developers unfamiliar with the code to quickly become comfortable with the intended code behavior.

Practice: Functional testing

The team performs *functional testing* to ensure the functional behavior of the product (or system) corresponds to its specifications; this involves testing one component at a time. Unit testing focuses on testing the smallest individual units or components of the build, application modification, or system to verify that each component is built to design specifications. Functional testing is performed after the component has been unit tested to verify functionality against the requirements and specifications and before system and integration testing.

Practice: Regression testing

The team performs *regression testing* to isolate and resolve any errors or defects introduced during modifications based on change requests. This testing verifies that the component still meets its specified requirements and no adverse effects have been identified as a result of implemented changes. Regression testing focuses on executing test scripts and/or scenarios in functional and nonfunctional areas of a system after changes have been made, ensuring the product or system still meets its specified requirements and will not fail when deployed to the operational environment or adversely affect code currently in production. The agile team should conduct a degree of regression testing at the end of each level of testing to ensure any defects corrected during each testing segment have not caused additional defects. Over time, we encourage all of our teams to strive for automated regression testing to the greatest degree possible.

⊘ *< Security and QA happens at the end >*

The security and QA groups aren't on the team, so you exclude them from the sprint plans. Changing your approach to committing work can sometimes mean bringing in groups who were previously a different department. It's easy to forget about the QA or security people if they are not actively part of your teams. Leaving security and QA until the end equates to phases in traditional waterfall. If something is discovered in security or QA, the story must go back to the team for rework. At this point, the team has already moved onto new stories and must stop to accomplish the rework. The discovery may affect more than one story, which results in extra effort to fix the problems. If security and QA can be included throughout the lifecycle, problems can be uncovered early before requiring much rework. This also promotes collaboration and a shared commitment to security and quality as one team delivering solutions.

Practice: Automated testing

Automated testing must be considered part of the software development cycle. The tests themselves are an integral part of software development because they help minimize time spent debugging and help the team identify and resolve issues before users do. Prior to committing code, the code should be thoroughly subjected to automated test, and those tests should be committed with the code. This helps team members ensure their work is compatible with the code being committed. The entire automated test suite should be run against all code changes before that code is committed to ensure there are no conflicts with other areas of the project. When a bug is found in the system, the developer committed to fixing the bug should perform the following steps:

1. Write a unit test that expects the specific failing behavior not to occur.
2. Run the test, which should fail because the bug will still be in the code.
3. Fix the bug.
4. Run the unit test to ensure the test now passes.

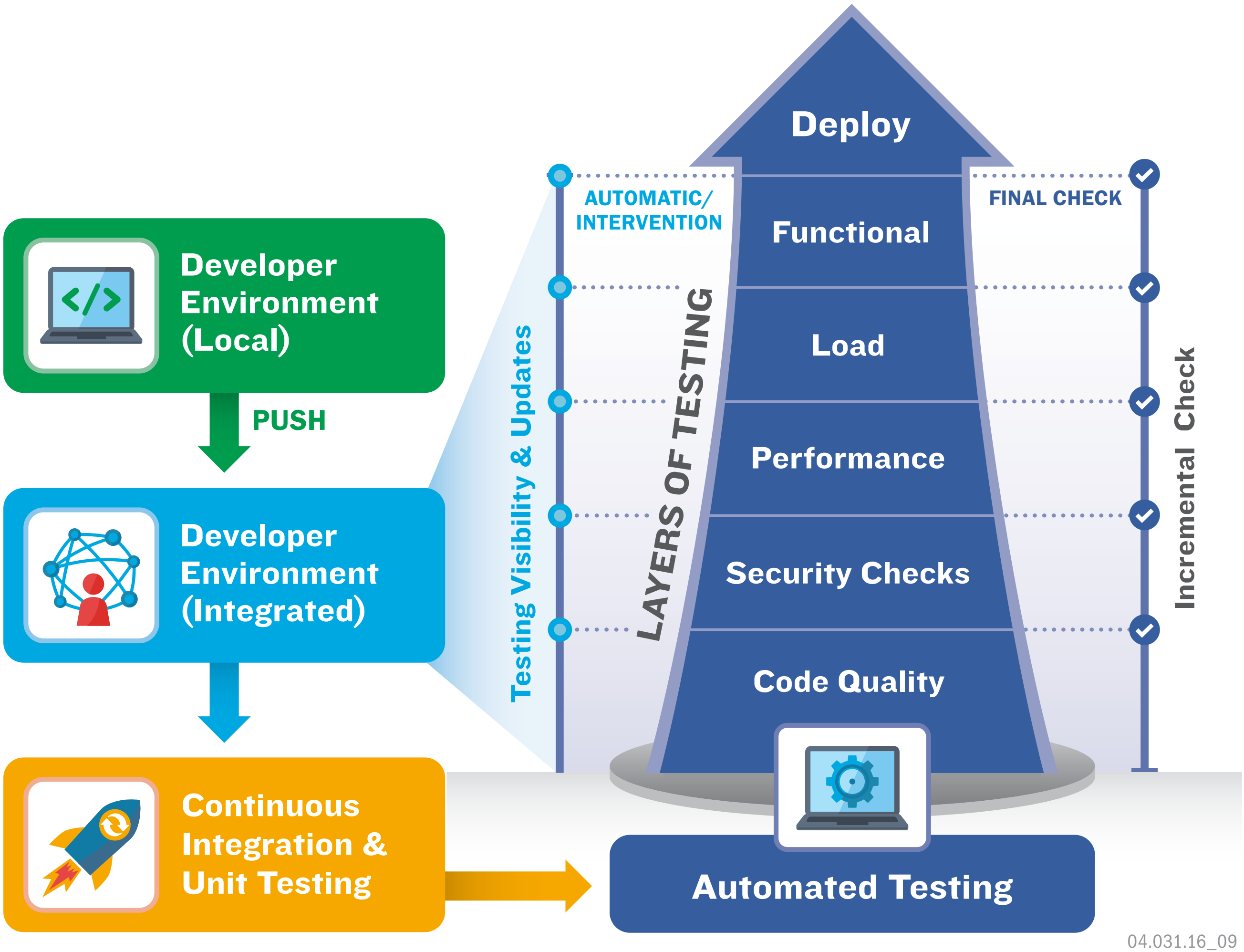
This practice ensures the bug can never be reintroduced into the system without being caught by the automated test suite.

Practice: User feedback and acceptance testing

Successful agile projects regularly put new software in front of the users for immediate feedback. This does not require a deployment to a production server but does require a demonstration or test server to which users have access and can try out new features still in development. Making this environment available increases communication between the users and the agile team. If features are found to be off track, the developers can start over, with only the loss of a sprint's worth of work versus 6 months or more if discovered later. User testing has the added benefit of showing users that while you may not have had a deployment recently, you are making progress on the highest priority request.

Play: Automate as much as you can

Figure 9: From development to deployment



04.031.16_09
Booz Allen Hamilton

Practice: Use continuous integration

Continuous integration is a foundational agile technical practice. It requires each team member to integrate their latest work with the trunk frequently—at least daily—and to have each integration verified by an automated build (with automated testing included). Continuous integration increases the quality of the software by reducing the defect escape rate and decreases maintenance and sustainment costs. Developers working from a local copy for days at a time is a bad practice and contributes to risky, complicated merges; continuous integration mitigates this risk.

Continuous integration means that unit tests are executed every time automated test builds are generated from the code repository. If one of the unit tests should fail during execution, the continuous integration mechanism notifies the project team every time it tries to execute the test, until the problem is resolved.

Continuous integration tools can be set up to notify the development team of any failed build. These enable the team to resolve small integration issues early, before they become large or multiple issues just prior to a release or deployment.

When combined with automated testing, code quality inspection tools, and vulnerability scanning tools, continuous integration becomes an even more powerful tool for the agile team. Automated unit tests identify problems early and prevent integration defects from piling up—reducing risk to release candidates and product deliveries. Automated code quality inspection tools and vulnerability scanning tools can identify poor coding practices, code violations, and security violations without the need for additional developer code review. Continuous integration increases code quality and reduces bugs in the deployed software.

Teams that practice continuous integration tend to be able to complete a build easily and have a lower cost of change than teams that have to manually exercise their build process.

Guidelines for successful continuous integration include:

- Providing the ability for a "one-click-build"
- Executing automated builds at every commit or at least once daily.
- Achieving 60% unit test code coverage as a target, with anything above 70% considered exceptional
- Providing automated notification to the entire development team when a build or unit test fails

Practice: Continuous delivery and continuous deployment

When a team has continuous integration working well, it can consider the more mature practices of continuous delivery and continuous deployment.

Continuous integration gets us as far as having a server somewhere that builds the software and runs the automated tests, but it doesn't get it off that build server. Continuous delivery is that next step: ensuring each change to the software is packaged up and releasable, and we can push it to production with click. This means we must have automated and smoothed all the mechanical steps of the release and deployment process, which often slows down teams. In a continuous delivery environment, releases become boring.

Continuous deployment goes one more level. Each change is pushed to production automatically. You must be practicing continuous delivery before you can consider continuous deployment. To make this leap, all the qualitative and judgment aspects of releases that might have been performed by humans before are now automated. Do we really trust the automated QA? Have we covered the edge cases? Does this release degrade performance anywhere? Is the timing right for the business? When the software can automatically stop releases that should not go out but lets the others pass, we've reached continuous deployment.

Practice: Continuous monitoring

To have success with continuous delivery and continuous deployment, you must have the backbone of continuous monitoring in place. Without the ability to know how the application is performing, the processes in place no longer stand.

Continuous monitoring is constant validation that the application is functioning and performing as expected. Through the creation of monitoring tools, teams are able to remain confident that the applications are performing and functioning at their optimal levels. This means that we must know there is a failure before one occurs and build feedback mechanisms to gather information from the system.

When gathering information, it is important to understand:

- How and why is early detection of defects important to your project?
- What is the system availability requirement and what is the plan to achieve it?
- What is the labor cost of your operations?
- How many systems engineers, systems administrators, and database administrators do you have monitoring the production environments, and how do they react when there is an issue? (DevOps Playbook publishing soon)

When gathering information, it is important to understand: When keeping these questions in mind with continuous monitoring, teams are able to find issues and defects early and automatically, which in turn increases overall confidence and reliability in the application.

Practice: DevOps

DevOps applies a software mindset to how systems get deployed and maintained. Similar to agile itself, DevOps is more of a mindset with principles than it is any particular set of practices. A common way to describe it is with the acronym CAMS: Culture, Automation, Monitoring, and Sharing.
These principles dovetail with the overall agile principles while expanding the scope to include anyone involved in releasing, delivering, and maintaining software and its infrastructure.

The aims of a DevOps implementation are to:

1. Reduce risk
2. Increase velocity
3. Improve quality

Although it addresses a fundamental need, DevOps is not a simple solution to master. By integrating software developers, quality control, security engineers, and IT operations, DevOps can provide a platform for new software or program fixes to be deployed into production as quickly as they are coded and tested. That's the idea, anyway—but it is easier said than done.

To excel at DevOps, teams and organizations must do the following:

- Transform their cultures
- Automate legacy processes
- Design contracts to enable integration of operations and development
- Collaborate and then collaborate some more
- Honestly assess performance
- Reinvent software delivery strategies based on lessons learned and project requirements

Specifics on implementing DevOps can be found in the Booz Allen DevOps Playbook.

Practice: Seek an agile architecture

An architecture that supports agile projects must support the current effort and any future complexities. Loosely coupled service-oriented architectures are often used to leave room for future changes. Advancements in technology, such as microservices and containerization, affect the nature of the architecture and change the need to a virtualized infrastructure. Many projects do not have the luxury of starting new and need the legacy infrastructure and construct integrated. When these complexities arise, tailored solutions that take into account latency or potential performance issues must be developed with the end-user experience in mind. Architecture, just like any other part of the development, is work that must be planned, executed, and accounted for.

The more complex the solution, the more planning is needed for infrastructure, as well as application architecture. Using an architectural roadmap to plot out future needs will provide a type of scaffolding that will then be completed with a detailed design that builds out just enough infrastructure or detail needed to complete features. Depending on the size of architecture being built, one or several architectural epics are executed through technical spikes within the sprints. The level of documentation and diagrams needed will be informed by the agile principles in mind during development and implementation.

MEASUREMENT

Measurement affects the entire team. It is an essential aspect of planning, committing, communicating, improving, and, most importantly, delivering.

Play: Make educated guesses about the size of your work

It is important to understand what you are committing to and acknowledge when you do not. Agile teams seek to discover this understanding through the process of estimation.

One of the historical challenges of software estimation is the perception of precision. When a feature is estimated as “273 FTE hours,” it unintentionally gives a very false sense of precision and confidence in that estimate. In reality, a manager asked five different people to provide estimates across 150 requirements and summed them up.

Agile teams apply several simple practices to drive conversation, quickly expose what they collectively understand or don’t understand, and get some numbers that are approximately right instead of precisely wrong.

Practice: Use relative estimation

Relative estimation is a concept that those new to the agile world often struggle with, but it simply means “compare two things to each other.” If you’ve ever said something like, “Hey, this tree is twice as tall as that tree,” you already know how to do it.

When we think about estimating software, we recommend agile teams do so in terms of size. Software size is intended to capture, all at once, the:

1. Amount of work
2. Difficulty of the work
3. Risk inherent in the work

When paired with relative estimation, it results in questions like, Is this feature as complicated as the other feature we built last week? Or, If we take on this feature, is it more risky than this other feature? When work items are similar enough across these questions, we give them the same number on a scale (see table below) that the team has previously selected. As work items differ, the team discusses those differences to understand just how different the work is, relatively, and give a corresponding number from that same scale.

Table 4: Example relative estimation scales

Name	Examples
Modified Fibonacci	1, 2, 3, 5, 8, 13, 20, 40, 100
Small, not too big	1, 2, 3, 100
Same-sized work	1, too big, no clue
Powers of 2	1, 2, 4, 8, 16, 32
T-shirt sizes	XS, S, M, L, XL

These are just a few examples of scales we've seen teams use at Booz Allen. The most popular is the Modified Fibonacci scale because it was documented in a fantastic book (Agile Estimating and Planning by Mike Cohn) and is sold on estimation card decks [Cohn 2005]. But, its popularity is for good reason: The nonlinear sequence works well because the gaps reflect the greater levels of uncertainty associated with estimates for bigger, more unknown pieces of work. More mature and disciplined teams often move toward simpler scales, such as small, not too big (SNTB) and same-sized work (SSW); in these cases, they essentially seek to know if work is small and understood, or if more discussion is required before commitment. Don't try to look up SNTB or SSW; we made those names up.

What do we call these numbers?

Generally, agile teams call these unit-less measures of software size "story points," or simply "points." We recommend that teams avoid the temptation to continue to estimate in hours, even when deliberately using relative scales, because the perception to stakeholders is often a higher degree of certainty than there truly is. At one point, "ideal hours" were offered as an alternative to story points. An ideal hour is a mythical hour where you can work, uninterrupted, with all the knowledge and resources at your fingertips to get the job done. In practice, we find this causes too much confusion among the team and stakeholders—someone inevitably confuses an ideal hour estimate with an actual hour from reality.

Practice: Estimate as a team

A generally accepted practice among the agile community is to have the team make estimates together. This practice allows the team to discover how well work is understood, encourages knowledge sharing, builds team cohesiveness, and fosters buy-in.

⊘ *< Collecting data without purpose >*

Any data collected should be intentional and with purpose to improve the current process. Collecting the same data because you always have is waste. Review what you do collect and how it improves the system.

One of the most popular techniques for having these team conversations to obtain estimates is Planning Poker [Cohn 2016]. The gist of it is that the team talks about its work in a structured (but fun) way. It is based on a popular expert-based method called "wide-band delphi," but we encourage you to provide chips and salsa.

There are other methods (e.g., Affinity Estimation, Magic Estimation), and more are created all the time. What do they all have in common? They get the team to make the estimates together, they drive conversation, and they expose uncertainty. And numbers pop out that are just good enough to move forward.

Play: Use data to drive decisions and make improvements

Across many of our highest performing teams and programs, we've found a common love of data. The best software practitioners really are geeks about it. And why not?! Data is objective. Data tells a story. Data is awesome.

Agile teams should use data to drive important decisions, commitments, schedules, and improvements.

Practice: Use the past to predict the future

An important input to any planning process is the team's capacity for doing work. In traditional capacity planning, this may look a lot like the sum of your team's *planned* work hours during a given period. The trouble with this is you are making decisions solely on estimates and focusing entirely on the individual people on the team rather than the team itself.

In contrast, agile teams predict how much work they can collectively accomplish by continuously looking back at how much work they have accomplished. Yesterday's weather is the best predictor of today's weather (unless you live on the East Coast of the United States).

Velocity (and throughput)

For most agile teams practicing some variation of Scrum, they should measure their velocity. Velocity is the amount of work the team has historically been able to deliver in a single sprint. We usually recommend using three to seven sprints' worth of data to produce a rolling average of your velocity each time you walk into a planning meeting.

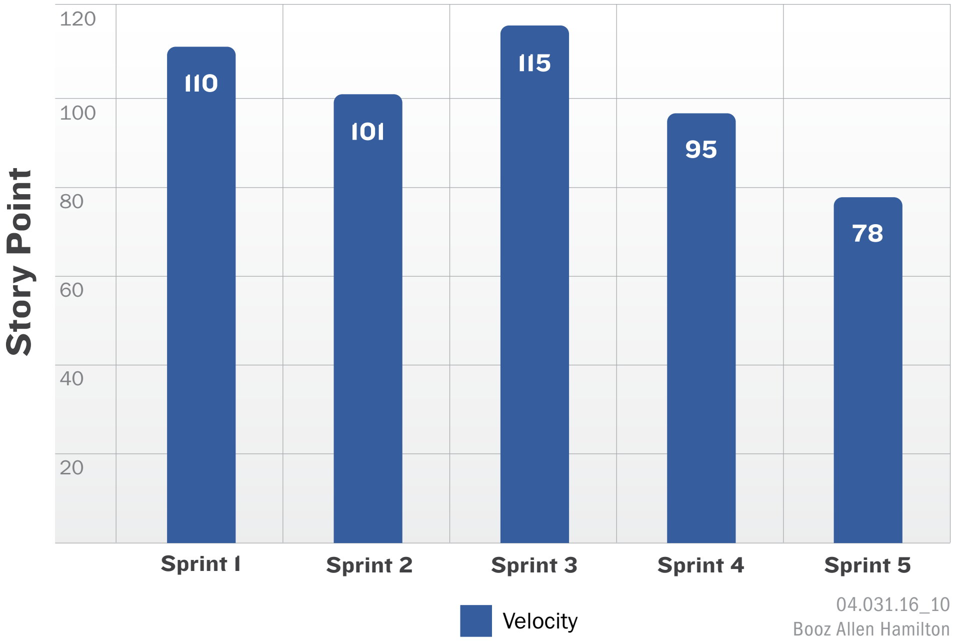
For agile teams that are moving toward a more Kanban-style delivery, the analog to velocity is throughput. Throughput is the amount of work the team is able to deliver over some period of time. A Kanban team may measure throughput in terms of hours or days.

⊘ < *Taking partial credit* >

Have you ever felt like you're super busy, but you're not getting many things finished? This happens to agile teams, too. It's tempting to quickly start all the current tasks on deck. But, code isn't really valuable until it's running and the user can try it. Since delivering value should be our primary measure of progress, our time is better spent taking things to done rather than starting more things. Taking partial credit for work does not help the team improve and deliver value. Inspect why the team is doing this; if there are blockers to completion, those blockers should be addressed rather than starting new work.

Some teams may get value in tracking their velocity in a Velocity Chart; see example below.

Figure 10: Velocity chart



04.031.16_10
Booz Allen Hamilton

Kanban metrics borrowed from lean manufacturing (cycle time, lead time, response time)

Kanban teams utilize several additional metrics from the manufacturing world because Kanban emphasizes the flow of work through the system to produce valuable outcomes. These metrics help understand how well the system is working.

Table 5: Kanban metrics and definitions

Metrics	Definition
Cycle Time	Average time per work item (the inverse of throughput—think about it!)
Lead Time	Average time it takes to deliver an item from the moment work starts
Response Time	Average time it takes to begin work on a single item from the moment it is added to the backlog

***Note:** These are "time per item" metrics. The average is calculated across a sample of your work items.*

Team predictability: Commitment variance

Commitment variance is a predictability measure of your team's ability to deliver on its commitments.

In essence, it is the percentage ratio of the team's delivered work to the team's committed work, averaged over time. For example, if a team commits to delivering 10 points in a sprint, but it actually delivers 12, the commitment variance for that sprint is 12/10*100 or 120%. The team has delivered 120% of its commitment.

If your team's commitment variance tends to be less than 100%, you may consider requiring the team to commit to no more than its previously delivered velocity instead of its rolling average velocity, for several sprints. This practice is literally called "Yesterday's Weather." Once the team becomes more predictable, you may consider returning to a rolling average velocity for planning.

⊘ *< Using velocity to compare across teams >*

Throughput and velocity are unique to every team. Said differently, you cannot compare teams based on their velocity measures. Relative estimates are completely dependent on the team that is doing the estimating; one team's 1 might be another team's 5. In practice, we've found that efforts to standardize estimation scales across multiple teams tend to diverge fairly quickly. Teams should be compared based on the value they are delivering per sprint and the predictability of their delivery.

Play: Radiate valuable data to the greatest extent possible

Agile teams are known for their **information radiators.** These are Big Visible Charts. Ron Jefferies said it best: "Display important project information not in some formal way, not on the web, not in PowerPoint, but in charts on the wall that no one can miss" [Jeffries 2004].

Certainly, some team environments pose a challenge in radiating information in this way. The trend toward globally distributed teams alone is an impediment. In these cases, the teams will have to explore other methods for radiating information and discover practices that work for them.

Practice: Burndown chart

Burndown charts are the go-to chart for agile teams. Simply stated, they track the amount of work the team has "left" on any given day, and—hopefully—the team's workload goes down most days. We have found burndown charts to be most effective for tracking progress "within a sprint."

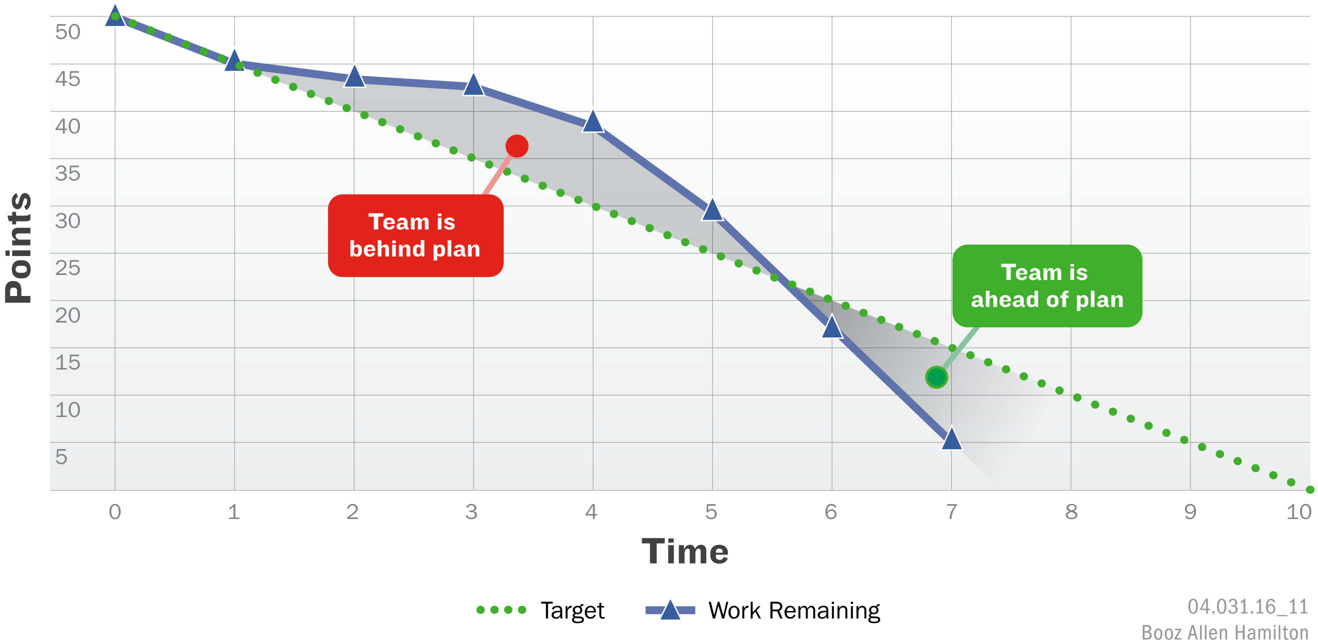
Coming out of a planning meeting (e.g., sprint planning), the team can sum up the estimates for all of the work in the sprint backlog. When stories are complete, according to the team's definition of done, points are burned down to show progress.

⊘ < *Extending the sprint* >

Often and inevitably, a challenge will arise where extending a single sprint will look like a plausible solution. And why not? For decades, we've grown accustomed to sliding schedules "to the right." Maybe the team did not complete all of the work they signed up to do, or maybe priorities drastically changed in the middle of the sprint. Or, even more simply, maybe there is a holiday in the sprint. The answer to this question should almost always be *No*. The team's sprint cadence is an essential piece of its rhythm and predictability. If you're not sure why the sprint is coming up short, that's a good thing to talk about at the retrospective.

In some cases, once a team gets started on the work, it learns a particular item is bigger than originally thought. That, too, is captured in the burndown chart as the points going up.

Figure 11: A burndown chart excels at tracking progress within a sprint



04.031.16_11
Booz Allen Hamilton

Note that for teams practicing Kanban, a burndown chart may not be the most useful way to observe progress. For that, we recommend a cumulative flow diagram (described below).

Burndown patterns

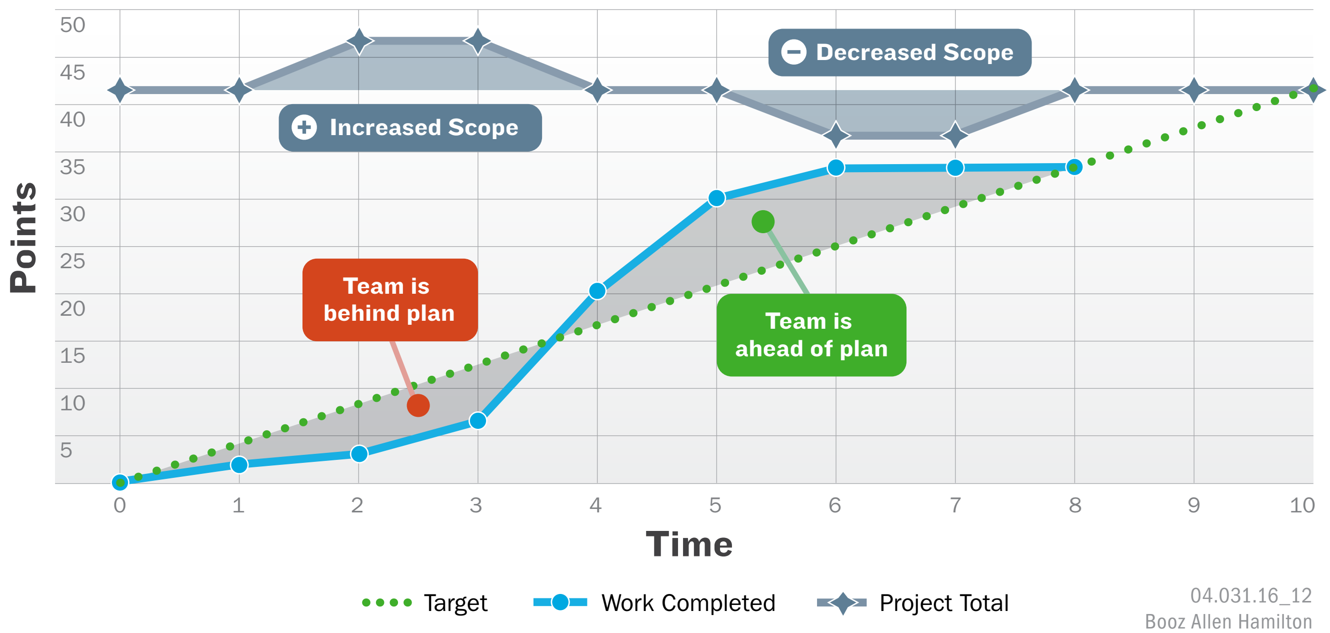
Keep an eye on the shape of your team's burndown charts. We're fans of this article from RallyDev, which speaks to this idea [CA Technologies 2016].

Practice: Burnup chart

Burnup charts are quite similar to burndown charts; they just go in the opposite direction. Whereas the burndown chart shows the team "burning down" their work to 0, the burnup chart shows work being completed and moving up. Likely the largest difference between the two is the addition of a target line in the burnup chart. As the team progresses, burning up progress, the total amount of work it expects to get done also is graphed. This chart is an excellent way to display changes in scope or understanding over time.

It is for this reason that we recommend teams use burnup charts for tracking bigger efforts across multiple sprints, such as for a large epic or feature delivery, or a multi-sprint plan time horizon (sometimes called a "release" or a "product increment").

Figure 12: A burnup chart is great for tracking progress across multiple sprints



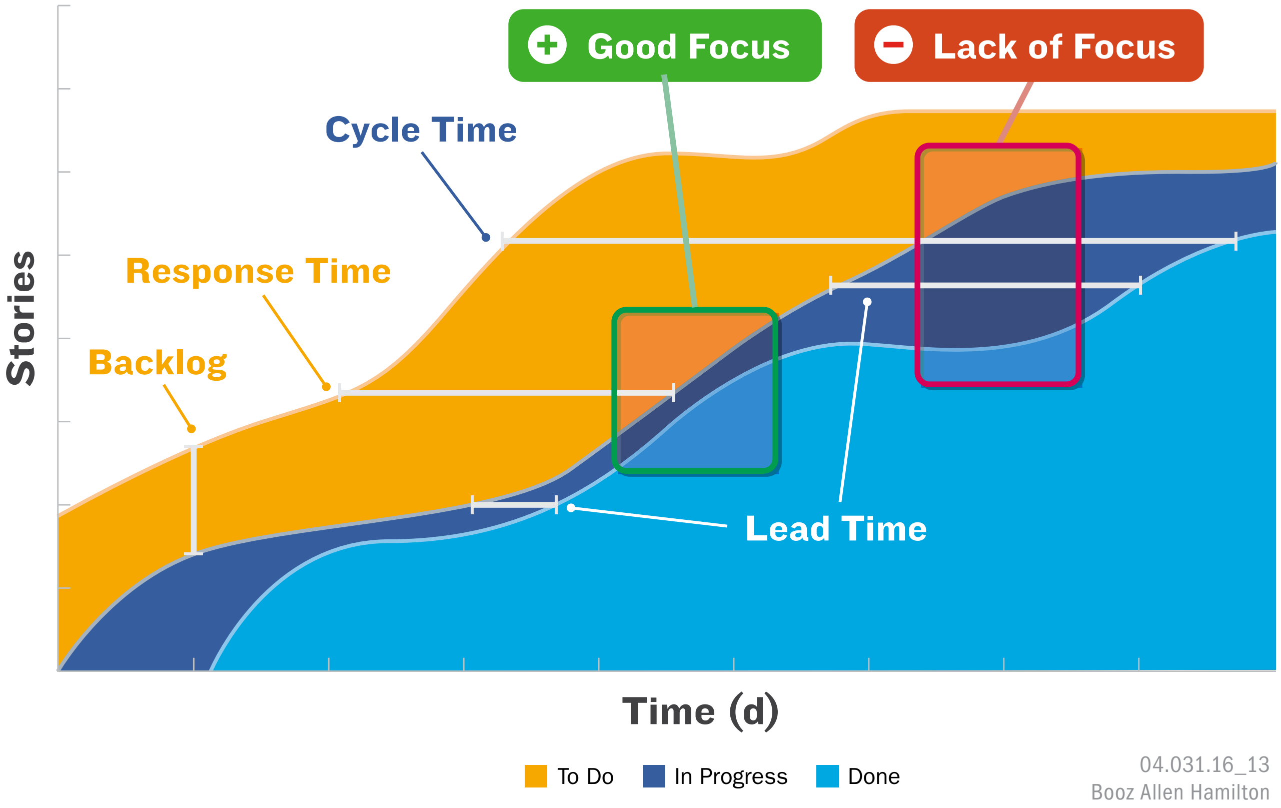
04.031.16_12
Booz Allen Hamilton

Practice: Visualize work in process using a cumulative flow diagram

Cumulative flow diagrams show where work is in your process. Over time, you can see how much work you have in any given stage of your product flow. This diagram is an excellent tool for teams to visualize their WIP. It is easy to observe bottlenecks, understand your team's level of focus, and get a sense of some of the Kanban metrics described previously (cycle time, response time, lead time).

For teams of all shapes and sizes, WIP should be managed and kept to a minimum to ensure continuous flow of value and reduce wasted effort.

Figure 13: A cumulative flow diagram shows a lot of information in a single picture



04.031.16_13
Booz Allen Hamilton

Play: Working software as the primary measure of progress

> *Working software is the primary measure of progress.*

Yes! In any discussion of measurement, it is essential that we not lose sight of an agile team's true measure: the regular and continuous delivery of working, high-quality software that is potentially shippable.

Practice: Technical debt

Technical debt is a useful metaphor for "stuff that will come back to bite you in the long term." Technical debt is the cost of fixing the structural quality violations that, if left unfixed, put the business at serious risk. The data on technical debt provides an objective frame of reference for the development team. It also provides a way for the development and management teams to have a tradeoffs discussion. Technical debt could be defects left unresolved, code that is not reviewed or unit tested, or shortcut architectural decisions. What technical debt truly means to your team is a worthy discussion to have often.

Whatever you determine to be your technical debt, you should track it. You are going to have to pay it off sometime. And, like interest, the cost of change increases over time as your technical debt ages and grows.

Here is a short list of code quality indicators that are worth tracking:

- Unresolved defects over time, sliced by severity
- Unit, integration, and functional test code coverage
- Frequency and duration of builds
- Code review coverage
- Code coupling and cohesion metrics

Practice: Value predictability

We've previously discussed velocity as a measure of the team's capacity to deliver work. Because of its name, we sometimes forget that it is slightly different than the velocity of the physical sciences (in physics, velocity is a measure of speed and direction). In the agile world, velocity has no sense of the team's direction.

Velocity is valuable input for planning, but it is not exactly an indicator of progress. To help get a sense of the team's direction and the value of its output, we turn to the value predictability measure. This measure is a comparison of the actual value delivered and the planned value delivered.

Coming out of a planning session, the team's Product Owner assigns a value estimate to each work item (usually on a simple relative scale, like 1 to 10). Then, coming out of a review, the team's Product Owner identifies how much value the team delivered for each item on the same scale. Both of these sets of values are summed and tracked over time.

Note, this is a measure that we've seen used by teams at Booz Allen, but it is also quite similar to a measure popularized by SAFe: the program predictability measure. SAFe suggests that this type of metric could be useful at higher levels of an organization and recommends a predictability measure between 80% and 100% as being "predictability sufficient to run a business" [Scaled Agile, Inc. 2016a]. We see no reason to disagree.

Practice: Customer satisfaction

Customer satisfaction is perhaps the most difficult part of any development effort to define and measure. It weaves together multiple factors, including user experience, value delivered, performance, timeliness, and more. Because of these complexities, it's important to engage your target groups regularly for direct feedback and collaboration.

The simplest approach we've seen teams successfully use to track customer satisfaction is to ask for a letter grade (e.g., A, B, C, D, E) coming out of all demonstrations or deliveries. Tracking these letter grades over time can give you a really solid sense of how successful the program is at delivering the value it says it will deliver.

One caveat: In cases where customers may churn a lot on direction, there is a slight risk that the customer satisfaction metric could be low in spite of the team's best efforts.

While direct stakeholder engagement is often more time consuming, it's necessary to develop a complete and accurate picture of your customer satisfaction. Some common strategies include regular engagement with a stakeholder committee or champion group, 1:1 and small group interviews, and user surveys. To avoid over taxing your stakeholders, develop a sampling strategy that engages different users based on your release plan over time. The combination of qualitative and quantitative data will facilitate alignment of your product with evolving user needs and business priorities.

MANAGEMENT

> “ ***Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*** ”

In a system with self-organizing teams, where is the room for a manager? What does agile management versus leadership look like and how does this affect everyone on the team? We take a look at how the manager leads in an agile organization.

As managers with agile teams, we may find that our role has evolved, but it is still valuable. With self-organized teams, there is always this question: What do project or program managers do now? Not only do the managers themselves ask the question, but also those who report to them. The traditional hierarchy is blown up and things may get uncomfortable at first. Here we present a few shifts to consider as you engage with agile teams.

The following plays represent a progression of skill sets a manager may go through. However, each play can also be considered a stance the manager may take in order to meet the need of the team(s) regardless of progression. As a manager, ask yourself, What type of manager do I need to be in order to best serve my team(s)?

Play: Manager as facilitator

Some managers will have great success looking at themselves as a delivery facilitator. How can you make delivery easier? What roadblocks can you remove? Who can you connect to quicken the pace of the whole team?

The facilitator role can be difficult for those who are more comfortable with assigning work (or having it assigned) and receiving status until it is complete. Agile tools and reports are designed to make status seamless and available to anyone who wants it, without having to ask each other. We are hoping the team members will keep each other accountable for fulfilling their own commitments, but the facilitator still is watchful and plays a guiding hand in these interactions. He or she will still make sure the right people are talking and encourage the team members to stretch themselves.

Facilitators help everyone hold their time as valuable. They take an active role in making sure meetings are valuable, are not too long, and have the right audience in the room. So, while it’s really the broad team that develops the sprint or release plan, facilitators make sure the meeting is set up for success, with the right information and the right players. Facilitators also help the daily standup move briskly and make sure any monitoring and reporting connections are being made to sponsors and executives.

Facilitators may find that they get more opportunity to focus on the people they work with—and encourage the team members to think in that frame as well. Formerly repetitive processes associated with status are now valued less than the cohesiveness of the team and building something valuable together. Facilitators ensure time is spent reflecting and discussing how the team works together (most focused during retrospectives but welcome at any time).

Play: Manager as servant leader

Through facilitative leadership, where you no longer need to direct the team but rather enable its own direction, you are ready to embrace servant leadership. In facilitation, you often learn you are not the person with the answer; rather, you look to the team for answers as it helps the individuals and team more than it helps you. And, teams are very wise. This can be a very humbling experience. The humility you learn as a facilitator guides your growth as a servant leader. You celebrate the team's success of producing a valuable quality product. You view yourself as the person always thinking of ways to improve team members' work-life to create their best work, by clearing impediments and not interfering. This is strange and different for those used to being the go-to person, or being a hub.

For teams used to having a clear leader to make all decisions, servant leadership can be very unnerving. Sometimes there are those who need permission to proceed. Moving to this leadership style can initially cause apprehension and delays with people who usually very efficiently complete tasks. Once again, managers facilitate an individual's reliance on the rest of the team. Agile practice provides us tools to help individuals answer their own questions. If you are ready for more work, turn to the Kanban board for the next prioritized item on the backlog. If you are unsure about roles, refer back to the team charter. If there is anything not covered in the agile tools, talk about it and create something for yourselves (e.g., knowledge base, wiki pages).

Play: Manager as coach

So you have been practicing agile for a while, and you find yourself telling stories about how your past programs handled situations. Your agile books are dog-eared and you've seen a lot of success and failure. Although you may hold the same role as a "Project Manager," you have more depth to draw from and are taking more of a coaching stance with your team. You've found that you ask a lot more questions than give advice, and you've taken a deep interest in developing others to unlock their personal potential.

Just as a team starts with a more prescriptive framework, a facilitator starts with training and later is able to apply lessons informed by personal experience. Experience with people, process, and technology coalesce for someone ready to coach. A coach knows when to apply skills from mentoring, teaching, and facilitating; coaches expect to adapt rapidly and help teams do the same.

The roles may have changed, but ultimately striving for a long-term, sustainably happy work environment means living through the changes. Throughout the process, acknowledge with each other when things are uncomfortable, and examine if it's the change or if something is truly not working. Issues may emerge that have always existed in the organization, so be ready to address them through organizational, process, or change reengineering.

ADAPTATION

> ***“At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.”***

Adaptation is key to improving team outcomes and adjusting as the environment and team change. In this section, we look at a couple of ways to regularly examine and find ways to improve the team and product.

Routinely adapting—and expecting adaptation—are vital to a healthy agile project and a healthy agile team.

Play: Reflect on how the team works together

If a team must pick only one agile practice to adopt, it should be retrospectives! As a way to inspect and adapt the way the team works together, they become a vital part of a healthy team. They take the form of a regular meeting for a team to be able to pause and reflect on how things are going, adjust process, and reflect on areas of improvement with each other. The simplest form of a retrospective would be to come together and ask a team questions like, What are we doing well? What aren’t we doing well? What would we change for the next few weeks? Retrospectives should be common enough that the team is comfortable experimenting with things. It is an opportunity to tune processes and behaviors. In addition to improving the work, it build trust and cohesiveness in the team. Many teams fall into a trap of skipping retrospectives when pressure is on. Don’t do it! Treat retrospectives with importance, and reach out to another facilitator or a coach if you’re having trouble finding value in the meeting. Holding a retrospective about once per month is a healthy practice for most teams.

⊘ < Under pressure, so skip the retrospective >

When a team feels it is behind on work committed, or unexpected issues arise that take time, its easy to think the retrospective is too long a time commitment. Sometimes you have to slow down to go faster. The retrospective is an investment in team. Without the opportunity to reflect in an allotted time, the team will continue to push aside issues that are affecting velocity. The mindset that it is an investment will help the team ultimately become more effective.

We always recommend Agile Retrospectives [Derby and Larsen, 2006] to increase your skills in conducting retrospectives. But there is no better learning than by doing.

Play: Take an empirical view

Be ready for experiments! We are engaged in creative knowledge work that did not exist yesterday. We can't expect to have all the answers. But we can expect to design smart experiments, try things, and learn. Apply this principle in all you do.

Whether you're deciding what technology framework to use, or you're just trying to spend less time on help desk tickets, consider an empirical view. Reach back to elementary school and borrow from the scientific method basics:

- Define the problem.
- Form a hypothesis.
- Define the resources you'll need from your team.
- Conduct an experiment—and experiment with only one variable at a time.
- Gather data—compile test data once completed.
- Consider what you learned. Take action on findings.

Keep an open mind about experiments to conduct; they can include technical, interface, process, or people.

MEETINGS

> ***Business people and developers must work together daily throughout the project.***
>
> ***The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.***

Play: Have *valuable* meetings

Effective meetings bring value to a team in the form of common understanding. The richest form of communication is face-to-face communications while drawing on a whiteboard to express ideas beyond words. Codified meetings are a central element in implementing agile, and they serve several purposes:

- The team gets to know each other and builds respect.
- Ideas can be expressed, built upon, and finalized—facilitating a shorter learning cycle.
- Meetings reinforce face-to-face communication, or at the very least talking.
- They give regular check-in points and a rhythm to work.
- They reduce the need for documentation to be passed around as a primary form of communication.

Effective facilitation of meetings

- Plan as far in advance as possible. Regular meetings should be on everyone's radar far in advance.
- Emphasize the importance of the meeting as part of work.
- Ensure everyone feels comfortable contributing by asking opinions and giving everyone a chance to talk.
- Timebox and keep to the timebox. Adding time to the planned meeting just contributes to the feeling that meetings are not useful.
- Stick to the agenda. Use tools like a parking lot to determine follow-on meetings.
- Provide notes for the meeting in an easily accessible place so decisions are not lost.
- Meet in the same team space as much as possible. Keep things like your sprint Kanban board there and refer to it during meetings.
- Emphasize respect for each other. Only one person should speak at a time.
- Build working agreements. Ensure a working agreement for meetings is discussed for each type of meeting.
- Review actions from previous meetings to ensure team members are keeping commitments and get the assistance they need.

The following table provides an overview of the types of meetings the agile team can employ during its project's lifecycle.

Table 6: Useful, possibly valuable meetings

Meeting	Outputs	Frequency
Release Planning	Scope out several sprints worth of work, consider risks, and high level design	Once per Release (may occur quarterly)
Sprint Planning	Commit to the next few weeks of work, with whole-team buy-in	Once per Sprint
Backlog Grooming	Be sure the backlog is emergent and well-prioritized; refine detail and cancel dead stories	Once per Sprint
Sprint Review	Show the work that's completed and receive regular feedback on the product	Once per Sprint
Retrospective	Take time to reflect as a team on how we're working, and how we can improve ourselves or our process	At the End of Each Sprint
Daily Standup	Coordinate the day's work, find impediments, and hold each other accountable	Daily

AGILE AT SCALE

We know that agile teams can better handle changing priorities while being more productive and more predictable [VersionOne 2016]. But, the same things that help us create that environment on a team of 10 may break down when we have 50, 80, or 200 people on the team. Agile at scale is more than simply applying agile practices on multiple teams. Agile at scale requires multiple levels of coordination to ensure all the teams in the enterprise are moving in the same direction. Other aspects such as culture and funding must be considered as well. Implementing agile in an organization cannot be top down or grassroots alone. A successful adoption includes energy coming from both directions.

Keep in mind that, as in teams, agile is not sufficient as its own goal. Look to agility to find happier customers, higher team morale, more space for innovation—but not just because it's agile.

As you get to it, let's just acknowledge **this is hard.** There are no silver bullets, no singular unifying theory or framework that will guarantee stress-free success with agile at scale. It will take trying, learning, and re-learning. For an organization not accustomed to agile methods, it will take significant energy to transition. But, when in doubt, remember to **use agile to scale agile.** Build in constant experimentation and feedback to how you work. Build a foundation of great agile teams; develop empathic lean-agile leaders. Simple structures and simple changes are almost always better than complex ones. So **think simply,** even for a complex organization.

⊘ ***< Lack of an executive sponsor >***

The team is enthusiastic about being more collaborative and delivering more value to the end user, but does not see how "management" can help. Or, the management thinks agile is something only the software developers do. Supported by the annual findings in VersionOne's State of Agile survey [VersionOne 2016], most barriers to the successful adoption of sustainable agility derive from culture change and support (not just buy-in) from executive-level leadership. It is essential to have a leader who understands agile and has the authority to set the vision for its adoption across the organization, program, or team.

Much more so than in individual teams, **consistent training is essential.** Small organizations can accidentally run into great discoveries and organic ways of working, but this is much less likely at scale. Whether using a packaged delivery framework or your own creation, training and alignment are essential. It's important that everyone within an organization uses the same words, understands the same principles. But, don't overemphasize the process! **Process isn't everything.** Process is important, but it's insufficient for success. Culture is as or more important than process. As your organization transitions toward agile, be mindful that culture change is happening. Guide it, and treasure great culture.

Play: Train management to be agile

To migrate an entire organization to a new way of thinking and acting, all members of the organization need two-way communication to help everyone understand what it means to the organization and for their particular role. This is most crucial for management at all levels. By educating the management team in the changing mindset and overall vision for the organization, the team can work through the change management together with a unified approach. This communication assists with organizational change, as well as ensuring the teams have the resources and space they need to norm.

Play: Decentralize decision-making

The key to remaining agile with more than one team is allowing decisions that are made frequently or that lack widespread impact to be made at the team level. Requiring an organizational body to meet and make all decisions for all parts of a large organization often causes delay. The decision delay can be as long as it would take to execute the decision, causing waste in the form of time for an organization. Localized decision-making also contributes to team ownership, which is essential to embodying the agile principles. Centralized decision-making still has a place in large organizations, however. When deciding to centralize a decision, examine whether the decision is infrequent, impacts the entire organization, and has a time constraint. This will help guide whether it would benefit from decentralization.

Play: Make work and plans visible

Scaling agile development teams to program or portfolio levels means managing competing needs through alignment of vision and synchronization of sprints and delivery with dependence upon each other. Just as a team has a backlog that is regularly prioritized and elaborated, a program or portfolio must also have a backlog that is groomed to allow for prioritization of work. The backlog at this level needs to have flexibility to align with near-term organizational priorities and enough elaboration to assign the appropriate level of resources. A planning roadmap is a good tool to plan for the near term defined as the current fiscal quarter. Less detail in the roadmap is needed as it progresses to the future, as the longer term needs will continue to be prioritized and elaborated upon on a regular basis.

Play: Plan for uncertainty in a large organization

How do you plan for something unplanned? As discussed earlier, decentralizing decision authority is one technique to enable flow and eliminate waiting for a centralized authority. Other ways are to plan only as longterm as required. Put energy into only the immediate or funded activities. Planning beyond that should consume less effort so the team can pivot if organizational context changes. Planning horizons vary among organizations, but often fiscal schedules or contracts guide the overall roadmap. Agile is a mindset; even those at the highest level of the organization need to remember that.

⊘ *< Assuming resources, time, and scope can all remain fixed >*

We're all familiar with the iron triangle of time, resources, and scope. Agile turns it upside down and adjusts scope assuming time and resources are fixed. Time is fixed by using a cadence, and resources are usually associated with money, which is also fixed during a period of time. By fixing scope as well, you're assuming everything is known in advance and the delivery and solution are predictable. This is often not reality. Much of the solution must be discovered through creativity and experimentation. Fixing all three parameters results in date slip, cost overrun, and/or insufficient delivery. There is no room for injecting new, valuable items based on learning and discovery. By allowing the scope to flex and adjust to accommodate changes in the ecosystem or lessons learned through experimentation, we shift from being plan focused to being value focused.

Play: Where appropriate, use a known framework for agile at scale

Significant thought has been put into large-scale agile delivery frameworks. We acknowledge this remains an area of development and change. There's no need to start from scratch. In particular, we'd advise you to look at SAFe, LeSS, and Nexus. Consider these as a starting point. You will find that each is well documented by its founders, but all share a notion of being context-aware for your situation, and customizing to what makes sense for you, so long as it remains built on a solid foundation of agile principles and values.

Some known frameworks to explore:

- SAFe: http://www.scaledagileframework.com/
- LeSS: http://less.works
- Nexus: http://scrum.org

STORIES FROM THE GROUND

U.S. Army Training Program: An Agile Success Story

By the mid-2000s, the program that the U.S. Army used for training was coming apart at the seams.

By the mid-2000s, the program that the U.S. Army used for training was coming apart at the seams. More than a decade old at the time, the Automated Systems Approach to Training (ASAT) relied on old technology, and, despite a slew of add-ons, patches, and workarounds over the years, the program couldn't keep up with training needs, delivered inconsistent instruction, contained redundancies, and was expensive to maintain.

To replace ASAT, the Army decided to develop a new system, the Training and Doctrine Development Capability (TDDC), which would ostensibly be state of the art. This plan didn't quite work out as hoped. While the TDDC was designed to take advantage of the Web and of gains in hardware capabilities, the program's builders weren't as forward thinking in their methodologies. Structured primarily around traditional waterfall development techniques, the project continued for several years and chewed through nearly $100 million. However, the resulting program never worked to anyone's satisfaction. It lacked the basic functionality users wanted and it couldn't handle even a minimum number of concurrent training professionals.

Implementation delays would have meant users had to endure ASAT for a while longer, but the Army chose to scrap the TDDC entirely and replace it. This time, Army technologists were determined to try a different approach. A requirement of the newly drawn-up Request for Proposals for the new Training Development Capability (TDC) was that the contractor use agile methodology, with collaborative teams, frequent iterations, constant load testing, and deep engagement by the user community. A fully working product was completed by 2008, less than 2 years after the project's start—and there have been no hiccups. The system has been successfully rolled out to all of the Army training schools as a replacement for the ASAT system.

"Because of the first fiasco with TDDC, I came to the initial 30-day evaluation of TDC ready to fail it quickly and take an early flight home," says Henry Koelzer, a retired artillery NCO and early evaluator of the project. But after just a few hours, he decided, "This system, and the agile programming methodology, was going to work."

The primary failing of ASAT was its dependence on 1990s two-tiered, fat client architecture, which resulted in a wholly decentralized program. "Every school was a system in itself," says Dennis Baston, who is retired now but was a Supervisory Systems Analyst at the U.S. Army Training Support Center.

For example, the training software used at Fort Knox's armor school, Fort Benning's infantry school, and Fort Sill's field artillery school had to be loaded manually on servers at each of these locations. And because the applications were stove piped, the installations at the separate schools could not practically communicate with each other. The sheer redundancy of the courseware and the need to dedicate as many as 50 different servers exclusively to ASAT was a huge a drain on technology and financial resources.

What's more, once a course was placed on the server, individual trainers at each school could tweak it to their perceived needs. As a result, there were multiple versions of each set of training materials floating around, and no way of knowing which was the most current. In fact, sometimes a course got so lost in the system that it could only be found with an extensive search—and lots of manpower earmarked for it. At Fort Knox's armor school, for instance, after a search for the most current version of the weapons maintenance course, Army training professionals finally found it in the music school. "Who would have guessed those people were so hard core," Baston says.

Baston adds that when congressional investigators and U.S. prosecutors asked to see the training content related to interrogation methods used at Iraq's Abu Ghraib prison after military personnel were found to have abused inmates there in 2003, it was impossible to definitively decipher which version each soldier actually received.

Consequently, the Army's goal in developing TDDC (and later TDC) was to provide an integrated and centralized repository of training products that were approved, under development, or being considered for general use. In addition, secondary benefits sought included eliminating the duplicate content and reducing the time to develop training products.

The contract to build the TDDC was awarded to what Koelzer calls "a major company; one you would immediately recognize." Despite the vendor's reputation and resources, the waterfall approach doomed the project from the start. Following the typical waterfall techniques, program requirements were set in stone during the planning phase even before one line of code was written. No Army users—trainers or trainees—saw the interfaces and tested the functionality until TDDC was completed and delivered. "We gave them the use case, function points, and other major specifications, and when they were all done, they gave us the software, which was going to be a surprise, either good or bad," recalls Baston.

The contractor tried to minimize the risk of the waterfall method by pairing it with spiral development techniques, which involve more testing and even agile-like iterations during the project, but the spiral model shares a fatal flaw with the waterfall model: the program's requirements cannot change during development. So government evaluators were uncertain what they would see when the product was finally delivered. Combine that with the spiral approach of working on overlapping aspects of the project at the same time, with separate mini-development teams basing their activities on user requirements, functions, and features that were frozen in time during upfront planning. "So what you end up with is organized chaos," Baston says.

But even given that the waterfall method doesn't allow for modifications in project design, Baston says, "What was delivered didn't meet the requirements that were specified in the first place." He attributes this result to the fact that he and other evaluators could not see, or make corrections to, what was being produced until the very end.

For example, the system was supposed to support 6,000 training developers. But the software couldn't handle a load anywhere close to that—perhaps fewer than 100. Baston pins the blame on the contractor's testing process. Rather than assessing the system with real developers and realistic numbers of concurrent users, the contractor used a few of its own coders and not in sufficient numbers to push the software to the breaking point.

The outcome couldn't have been more of a disaster. After carefully evaluating the TDDC, Baston determined with 98% certainty that it could not be fixed and should be shelved. However, the project lead, a two-star general who was the deputy chief of staff for operations, was not willing to trash such an expensive effort even though Baston had given it a 2-percent chance of being fixable. Says Baston, "He wanted more certainty in our findings. So we had to go back and do more testing, more in-depth analysis, and we ended up with a 100% certainty that it was a complete, unrecoverable failure."

The project was then rebid, this time as an agile development effort. Phase 1 of the TDC, which began November 1, 2006, was a 30-day demonstration phase, at the end of which the prospective contractors had to demonstrate a prototype to a packed house of about 30 government evaluators. On the basis of this session, the contract was awarded to the contractor team of Unitech, Booz Allen Hamilton, and MPRI.

One immediate advantage of agile methodology over the waterfall approach was its continuous performance testing regime even during the development of the software. For example, load measurements were conducted each month with an application that estimated total system capability based on the behavior of the program when accessed by a large number of concurrent users—as many as 30,000 by the time the software was ready to launch. **In addition to merely issuing a "yea" or "nay" to do it at no cost. However, when it went beyond a small adjustment, the Army and the contractors negotiated ways to put more resources into that area of the system while streamlining other sections.**

For example, the military had failed to include a critical security function in its system requirements. When that substantial shortcoming became evident, the development team and the Government hammered out ways to make up for it, eventually agreeing to reuse some of the existing system accreditation documentation from the earlier programs. This freed up resources to tackle the security gap.

In the end the project came in on budget and on time. "What I saw happening was that there was an acceptance of the system from the user base as opposed to the contractor having to try to force its finished results on people," says Baston.

The final phase of the project, including deployment, maintenance, and data conversion, lasted from July 1, 2008, through September 30, 2008, when the training system went live.

The project was successfully deployed, and now training professionals can access courseware via a web browser and use the portions of it they need without corrupting the original program. As new content is added to the courseware templates, the system keeps track of which version is the most recent and who is responsible for it. When an appropriate supervisor signs off on a new version, the updated training materials are marked as complete and are made available to anyone with TDC access.

TDC has already generated numerous critical improvements with tangible gains. Perhaps the single largest benefit is TDC's impact on the preparation of course description documentation—known as Course Administrative Data (CAD) and Program of Instruction (POI)—which ultimately determines funding for training efforts. Accuracy is essential, so each CAD or POI undergoes a lengthy review by financial, training, and training development experts at Training Operations Management Authority

(TOMA) before submission to the Department of the Army. With ASAT, schools submitted these documents by exporting their databases to hard drives, which were then mailed to TOMA. In turn, TOMA personnel would import the data onto their servers, indicate necessary changes, and then send the edited documents back to the schools. The schools would make the required corrections, and the process would begin all over again. This system was so cumbersome that TOMA could barely meet the Army's minimum requirements for training assessment.

In sharp contrast, under TDC, CAD and POI are sent to TOMA through the workflow architecture within the system. TOMA receives notification electronically, and its experts then make comments directly into the files and route them back to the school for changes. As a result, submission times to the Army for CAD and POI have been reduced from about 1 month under ASAT to 1 day under TDC.

In addition, TDC's security architecture permits compartmentalization of information not possible under ASAT. With ASAT, restricting which information each user had access to was a complicated process. As a result, sometimes unauthorized users would inadvertently edit or change a file that didn't belong to them. By providing five separate domains, TDC allows supervisors to limit user access to only those programs they're authorized to work on. TDC also allows for consolidation of equipment, which reduces hardware, support and security costs, and complexity. ASAT ran on 78 different servers, each of which had to be housed in a restricted physical location. TDC runs on just a handful of web servers and a single database server.

Currently, TDC is used by almost 3,000 people on a daily basis and intermittently by an additional 3,000 users.

For more information

Shawn M. Faunce, faunce_shawn@bah.com, Booz Allen Hamilton

Dan Tucker, tucker_dan@bah.com, Booz Allen Hamilton

Haluk Saker, saker_haluk@bah.com, Booz Allen Hamilton

Wyatt Chaffee, chaffee_wyatt@bah.com, Booz Allen Hamilton

PARTING THOUGHTS

For teams new to the agile mindset, this Agile Playbook offers recommendations for achieving sustainable agility and success.

Use this advice along with your own conversations to instill team collaboration and improve delivery efficiency as you address and execute a vision.

We hope this gives you a good place to start. Keep in mind that each team can and should mold its approach, gain momentum, improve its skill, and become more mature.

For more information about Booz Allen, our Systems Delivery business, or our agile practice, please visit www.boozallen.com/systems-delivery or reach out to agile@bah.com.

ABOUT BOOZ ALLEN

Booz Allen Hamilton has been at the forefront of strategy and technology for more than 100 years.

Booz Allen Hamilton (NYSE: BAH) has been at the forefront of strategy and technology for more than 100 years. Today, the firm provides management and technology consulting and engineering services to leading Fortune 500 corporations, governments, and not-for-profits across the globe. Booz Allen partners with public and private sector clients to solve their most difficult challenges through a combination of consulting, analytics, mission operations, technology, systems delivery, cybersecurity, engineering, and innovation expertise.

With international headquarters in McLean, Virginia, the firm employs about 22,600 people globally and had revenue of $5.41 billion for the 12 months ended March 31, 2016. To learn more, visit www.boozallen.com.

About Booz Allen Systems Delivery

Booz Allen provides a full-stack enterprise systems delivery capability and provides high-performing, cross-functional agile teams.

We engage and lead across the full lifecyle: from user research and human-centered design through technical implementation and help-desk support. We employ over 3,500 software development staff across a variety of disciplines.

We approach agility with responsibility. Booz Allen maintains a current Capability Maturity Model Integration (CMMI) for Development (CMMI-DEV) Maturity Level 3 appraisal rating for its IT Team Development Organization. Initially achieved in September 2005 through an appraisal led by the Software Engineering Institute (SEI) (now known as the CMMI Institute), this rating was most recently refreshed in October 2014 by an external appraiser. The firm has completed 16 consecutive successful CMM/CMMI appraisals.

About Booz Allen's agile practice and experience

Facilitate adoption and implementation of the agile mindset and practices.

Booz Allen has applied agile practices for more than 12 years and currently has more than 150 active agile Systems Delivery projects. These projects range in size from small mobile application development efforts to complex enterprise solutions supported by more than 200 project team members.

The firm has invested heavily in training, tools, and resources to facilitate adoption and implementation of the agile mindset and practices. It has partnered with the International Consortium for Agile (ICAgile), Software Education Global, Scaled Agile Academy, and Agile Alliance. We have more than 350 certified Scrum Masters and Project Management Institute (PMI) Agile Certified Practitioners (PMI-ACP), more than 60 SAFe Agilists (SA), and more than 800 ICAgile Certified Professionals (ICP) trained in the agile mindset and practices. We are also host to a growing cadre of staff with more advanced agile certifications, including ICAgile Certified Professionals in Agile Team Facilitation (ICP-ATF, 21), Agile Coaching (ICP-ACC, 20), and Business Value Analysis (ICP-BVA, 22) and SAFe Program Consultants (SPC, 15). Booz Allen's SDUniversity program, established in 2015, launched with a Department of Agile Delivery, where our staff are able to obtain agile training and certifications across many of these disciplines. Our Agile Community of Practice (CoP) has been in place for nearly 8 years. With more than 1,100 members, it serves as a clearinghouse to share and discover new practices and to stay abreast of emerging trends in the domain. In 2016, the Agile CoP hosted its first Agile Day at Booz Allen internal conference, attended by over 130 Booz Allen practitioners and featuring talks from a dozen of the CoP's members. Finally, our SmartSuite environment is a set of firmwide enterprise tools, processes, and intellectual capital that can be tailored to support agile delivery in the areas of document management, team collaboration, continuous integration, requirements management, automated code reviews, testing, and defect tracking.

REFERENCES AND RECOMMENDED READING LIST

Lyssa Adkins. 2010. *Coaching agile teams: A companion for Scrum Masters, agile coaches, and project managers in transition*, United States: Addison-Wesley Educational Publishers.

Lyssa Adkins. 2015. *Developing an internal agile coaching capability: A cornerstone for sustainable organizational agility.* (November 2015). Retrieved April 29, 2016 from http://www.agilecoachinginstitute.com/wp-content/uploads/2015/11/Developing-an-Internal-Agile-Coaching-Capability.pdf

David J. Anderson. 2010. *Kanban: Successful evolutionary change in your software business*, United States: Blue Hole Press.

Kent Beck et al. 2001. *Manifesto for agile software development.* (February 2001). Retrieved April 27, 2016 from http://agilemanifesto.org

Booz Allen Hamilton. 2015a. *Booz Allen becomes Scaled Agile Framework® (SAFe) Gold Partner.* (October 2015). Retrieved May 9, 2016 from http://www.boozallen.com/media-center/press-releases/2015/10/booz-allen-becomes-scaled-agile-framework-gold-partner

Booz Allen Hamilton. 2015b. *Booz Allen Hamilton acquires software services business of SPARC, LLC.* (November 2015). Retrieved May 9, 2016 from http://www.boozallen.com/media-center/press-releases/2015/11/booz-allen-hamilton-acquires-software-services-business-of-sparc

CA Technologies. 2016. *Reading a burndown chart.* (2016). Retrieved April 23, 2016 from https://help.rallydev.com/reading-burndown-chart

Mike Cohn. 2005. *Agile estimating and planning* (Robert C. Martin Series) 5th ed., Upper Saddle River, NJ: Prentice Hall Professional Technical Reference.

Mike Cohn. 2016. *Planning poker: An agile estimating and planning technique.* (2016). Retrieved April 21, 2016 from https://www.mountaingoatsoftware.com/agile/planning-poker

Rachel Davies and Liz Sedley. 2009. *Agile coaching*, United States: The Pragmatic Programmers.

Esther Derby and Diana Larsen. 2006. *Agile retrospectives: Making good teams great*, United States: The Pragmatic Programmers.

ICAgile. 2010. *International Consortium for Agile (ICAgile).* (2010). Retrieved April 22, 2016 from http://www.icagile.com

Innovation Games. 2015. *Product box.* (2015). Retrieved April 23, 2016 from http://www.innovationgames.com/product-box/

Ron Jeffries. 2004. *Big visible charts.* (October 2004). Retrieved April 22, 2016 from http://ronjeffries.com/xprog/articles/bigvisiblecharts/

Ron Jeffries. 2015. *Um, agile software development requires software development.* (July 2015). Retrieved May 13, 2016 from http://ronjeffries.com/articles/015-jul/um-software-development/

Corey Ladas. 2010. *Scrum-ban.* (2010). Retrieved April 29, 2016 from http://leansoftwareengineering.com/ksse/scrum-ban/

Frederic Laloux. 2014. *Reinventing organizations: A guide to creating organizations inspired by the next stage of human consciousness*, France: Laoux (Frederic).

Jeff Patton. 2014. *User story mapping: Discover the whole story, build the right product*, United States: O'Reilly Media.

Pichler Consulting. 2016. *The product vision board.* (May 2016). Retrieved April 23, 2016 from http://www.romanpichler.com/tools/vision-board/

Rally Software Development Corporation. 2014. *Impact of agile quantified: Swapping intuition for insight.* (2014). Retrieved May 6, 2016 from https://www.rallydev.com/finally-get-real-data-about-benefits-adopting-agile

Scaled Agile, Inc. 2016a. *Metrics—SAFe.* (2016). Retrieved April 23, 2016 from http://www.scaledagileframework.com/metrics/#P2

Scaled Agile, Inc. 2016b. *Scaled Agile Framework—SAFe for lean software and system engineering.* (2016). Retrieved April 21, 2016 from http://www.scaledagileframework.com

Greg Smith and Ahmed Sidky. 2009. *Becoming agile—in an imperfect world*, Greenwich, CT: Manning Publications Company.

Jeff Sutherland. 2014. *Scrum: The art of doing twice the work in half the time*, United States: Crown Business.

Jeff Sutherland and Ken Schwaber. 2013. *ScrumGuides.org.* (July 2013). Retrieved April 23, 2016 from http://scrumguides.org

VersionOne. 2016. *State of agile report.* (March 2016). Retrieved April 23, 2016 from http://stateofagile.versionone.com